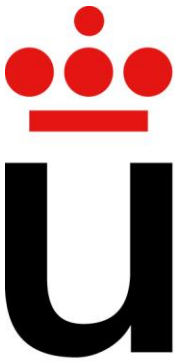


Universidad
Rey Juan Carlos

Sistemas Operativos

[PRÁCTICA 2 - MINISHELL]

BRAIS CABO FELPETE Y SERGIO PÉREZ SAMPEDRO

TABLA DE CONTENIDO

Autores

Brais Cabo Felpete

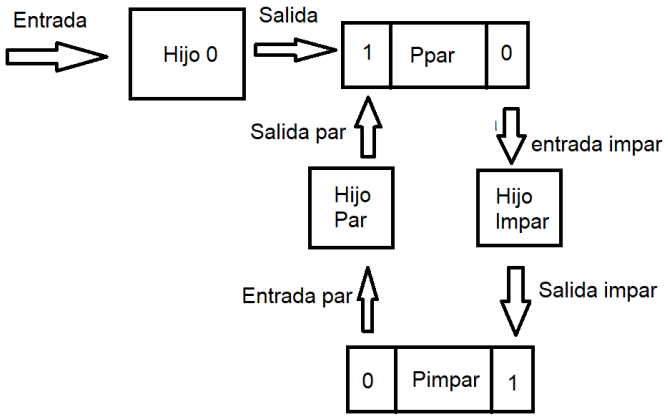
Sergio Pérez Sampedro

Descripción del Código

Diseño del Código

ESTRATÉGIA DE EJECUCIÓN DE MANDATOS Y GESTIÓN DE TUBERRÍAS

Para la implementación de los pipes con múltiples mandatos, hemos usado el sistema de 2 pipes. Para ello, diferenciábamos el primer proceso del resto. Ya que este nunca tenía que leer nada de pipe por entrada estándar. Para las siguientes n – 1 pipes, las hemos diferenciado si el número de comando que íbamos a ejecutar era par o impar. Por ejemplo, en ls | head -2 | wc, "ls" es el primer comando por lo que se ejecuta sin seguir el proceso del resto, "head -2" es el comando número 1 y "wc" es el comando número 2. Para conectar estos procesos, el padre iba abriendo los pipes que fueran necesarias. Para el primer proceso (número 0), si iba a haber más, abría el pipe "ppar" para que escribiese ahí su salida. Para los comandos impares abría el pipe "pimpar" para que el hijo escribiese ahí, y estaba abierta la ppar para que leyese de ahí su entrada estándar. Para los comandos pares, abría el pipe ppar para que escribiese ahí y leyese la entrada estándar de pimpar. Los hijos cerraban todos los pipes y el padre iba cerrando los pipes que no iba a necesitar el próximo hijo. Por ejemplo, en el caso de que el último hijo fuera impar, cerra la escritura en impar y ya que el próximo hijo iba a ser par y no iba a necesitar escribir en ese pipe. Cuando el comando fuera el último en ejecutarse, no se redirigía la salida estándar al pipe.



IMPLEMENTACIÓN DE BACKGROUND

Para implementar el background, no hemos hecho un waitpid al uso para que el usuario no tenga que esperar a la hora de seguir utilizando la Shell. El proceso o procesos siempre se van a estar ejecutando sin que intervenga en la interacción del usuario y cuando se ejecuta cualquier instrucción hacemos un waitpid con el flag WHOHANG para comprobar si los procesos se han terminado o no.

IMPLEMENTACIÓN DE SEÑALES

Para las señales, hemos contemplado 3 casos.

1. Se llamaba por teclado la señal cuando nada se estaba ejecutando. Para esto hemos reprogramado la señal para que se nos muestre un ^C y otra vez el prompt.
2. Se llamaba la señal en un proceso en background, ignoramos la señal.

3. Si se llama la señal mientras se están ejecutando procesos hijos, como es la señal por defecto, los mata y volvemos a mostrar el prompt.

IMPLEMENTACIÓN DE JOBS+FG

Para implementar el Jobs, hemos creado un array dinámico de structs. En cada struct guardábamos el pid de los hijos que habían sido mandados al background, el número de hijos, los números de los hijos que ya habían terminado con un array de booleanos y la línea que nos había mandado ejecutar el usuario. Cada vez que se pulsa la tecla "intro" se comprueba el estado de los struct del array. Si algún hijo ha terminado se actualiza el struct, si algún struct tiene el número de hijos que ha terminado igual a la cantidad de hijos, se muestra ese job y se indica que ha terminado. Si el usuario ejecuta el comando Jobs, se hace lo mismo, pero también se indica el estado de aquellos hijos que no han terminado. Para ejecutar el fg, si no se nos pasa el número de hijo, se hace un waitpid restrictivo y se espera activamente a todos los hijos asociados al último comando que se ha mandado al array de Jobs. Si se ejecuta el comando con un número, hace lo mismo, solo que en una posición del array concreta y desplaza los Jobs que estuvieran por detrás de el en el array.

Principales Funciones

	Main	Nombre	Tipo	Descripción
Variables Locales	Variable 1	**Buffer**	**Char[]**	**Array de char que sirve para leer lo que ha introducido el usuario.**
	Variable 2, 3, 4	**Wd, us, hostname**	**Char[]**	**Todas estas variables sirven para imprimir un prompt similar al de la Shell ya que guardan la ruta actual, el nombre del usuario, el nombre del host respectivamente.**
	Variable 5	**line**	**Tline ***	**Guarda la línea introducida por el usuario parseada.**
	Variable 6	**ljobs**	**Jobs ***	**Un array que guarda la información de los procesos que están en segundo plano.**
	Variable 7	**numero**	**int**	**Numero de comandos en segundo plano.**
	Variable 8	**mascara**	**mode_t**	**La máscara actual del sistema, al principio la inicializamos a (0022) que es la máscara por defecto de la Shell.**
Valor Devuelto			**int**	**Devuelve un entero indicando como ha**

				ido la ejecución y si ha habido algún error.
Descripción de la Función				**Esta función sirve para inicializar las variables que se van a usar durante la ejecución. También es la encargada de mostrar el prompt y de hacer la reprogramación inicial de la señal sigint. También es la que llama a las funciones que ejecutan comandos (tanto internos como externos) y la que lee lo que ha escrito el usuario.**

	fgCommand	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento 1	**com**	**tcommand**	**Parte del input del usuario parseado para realizar distintas funciones dependiendo de si se ha introducido una dirección o no.**
	Argumento 2	**ljobs**	**jobs ***	**Ed creada para almacenar las instrucciones de background.**
	Argumento 3	**numero**	**int**	**Cantidad de procesos en background.**
Variables Locales	Variable 1	**i**	**int**	**Variable utilizada para recorrer los procesos de background en un for.**
	Variable 2	**j**	**int**	**Variable utilizada en un for para actualizar el array de procesos en segundo plano.**
Descripción de la Función				**Esta función manda a foreground procesos que estaban en background.**

	cdCommand	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento 1	**com**	**tcommand ***	**Parte del input del usuario parseado para**

				realizar distintas funciones dependiendo de si se ha introducido una dirección o no.
Descripción de la Función				**La función cambia el directorio de trabajo actual, si no se le introducen parámetros al directorio home del usuario. Si se introduce 1 parámetro y es un directorio correcto se cambia el directorio de trabajo a ese directorio. En caso contrario da un error.**

	redirect	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento1	**in**	**string**	**Un puntero a un string con el nombre del fichero para la redirección de input.**
	Argumento2	**ou**	**string**	**Un puntero a un string con el nombre del fichero para la redirección de output.**
	Argumento3	**err**	**string**	**Un puntero a un string con el nombre del fichero para la redirección de error.**
	Argumento4	**c1**	**bool**	**Indica si es el último mandato del comando.**
Variables	Variable1	**fi**	**File**	**Si existe redirección será el fichero input.**
	Variable2	**fo**	**File**	**Si existe redirección será el fichero output.**
	Variable3	**fe**	**File**	**Si existe redirección será el fichero error.**
Descripción de la Función				**Redirecciona los ficheros donde el usuario ha solicitado depositar las entradas, salidas y errores.**

	prompt	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento1	**us**	**String**	**Nombre de login del usuario.**
	Argumento2	**wd**	**String**	**Directorio de trabajo actual.**

	Argumento3	**hostname**	**String**	**Hostname del usuario.**
Descripción de la Función				**La función imprime el prompt de la minishell en una gama de colores.**

	executeNComands	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento1	**line**	**tline**	**Guarda la línea introducida por el usuario parseada.**
	Argumento2	**lljobs**	**jobs ***	**Estructura de datos de los procesos que están en segundo plano.**
Variables	Variable1	**pid**	**Pid_t**	**Guarda el pid de los distintos procesos hijo.**
	Variable2	**ppar**	**Int []**	**Tubería usada para conectar los hijos. En esta tubería escribirán los procesos pares y leen de ella los impares.**
	Variable3	**pimpar**	**Int []**	**Tubería usada para conectar hijos. En esta tubería escribirán los procesos impares y leen de ella los pares.**
	Variable4	**j**	**int**	**Se guarda el número de comando que se está ejecutando.**
Descripción de la Función				**Por medio de pipes y procesos hijo realiza todos los mandatos solicitados en la línea.**

	exitCommand	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento 1	**numero**	**Int**	**Cantidad de procesos en background**
	Argumento 2	**ljobs**	**jobs ***	**Ed creada para almacenar las instrucciones de background**
Variables Locales	Variable 1	**i**	**Int**	**Número para recorrer todos los procesos en background**

	Variable 2	**j**	**Int**	**Número para recorrer todos las partes de cada proceso en background**
Descripción de la Función				**Cierra de manera estructurada los procesos de la Minishell para evitar que se cierren en cascada**

	umaskCommand	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento 1	**com**	**tcommand ***	**Parte del input del usuario parseado para realizar distintas funciones dependiendo de si se ha introducido una dirección o no.**
	Argumento 2	**mascara**	**Mode_t**	**Mascara actual.**
Variables Locales	Variable 1	**MascaraAux**	**Int**	**Numero para calcular la cantidad de 0 a escribir.**
	Variable 2	**numero**	**Int**	**Numero de 0 a imprimir.**
	Variable 3	**octal**	**int**	**El numero en octal de la máscara nueva**
Descripción de la Función				**Si no tiene argumentos mostrará con un formato correcto la mascará actual. si no cambiara la máscara por el argumento dado.**

	checkOctal	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento 1	**n**	**Char ***	**La cadena que queremos comprobar si es un número octal de 4 cifras o menos**
Variables Locales	Variable 1	**i**	**int**	**Se usa para recorrer el for.**
	Variable 2	**aux**	**int**	**Convierte el string a entero.**

Valor Devuelto			**bool**	**El resultado de la comprobación del número.**
Descripción de la Función				**Comprueba si una cadena que se le pasa por parámetro puede ser convertida a un número octal de 4 cifras o menos.**

	crlc	**Nombre**	**Tipo**	**Descripción**
Variables Locales	Variable 1, 2, 3	**Wd, us, hostname**	**Char[]**	**Todas estas variables sirven para imprimir un prompt similar al de la Shell ya que guardan la ruta actual, el nombre del usuario, el nombre del host respectivamente.**
Descripción de la Función				**Sirve para reprogramar la señal de ctrl + c. Se imprime un ^C \n y el prompt de nuevo.**

	Crlc2	**Nombre**	**Tipo**	**Descripción**
Descripción de la Función				**Sirve para reprogramar la señal de ctrl + c. Se imprime un ^C \n.**

	mostrarjobs	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento 1	**ljobs**	**jobs[]**	**Estructura de datos en la que están guardados los procesos en segundo plano.**
	Argumento 2	**numero**	**int**	**Numero de procesos en segundo plano actualmente.**
	Argumento 3	**control**	**int**	**Podemos indicarle al programa si queremos que se muestren aquellos procesos que se están ejecutando o solo los que ya han terminado**

Variables Locales	Variable 1, 2	**i, j**	**int**	**Sirven para recorrer los for que hay dentro de la función.**
	Variable 3	**p**	**Int**	**Sirve para guardar el índice de los cambios**
	Variable 4	**cont**	**Int**	**Se usa como contador**
	Variable 5	**cambio**	**Int []**	**Array de pid que han terminado**
Descripción de la Función				**Muestra los procesos que están en segundo plano. Dependiendo del valor de la variable "control" que se le pase, mostrará solo los que han terminado o los que han terminado y los que están en ejecución.**

Casos de Prueba

Para comprobar si la MINISHELL funcionaba como se esperaba, hemos ido probando los distintos apartados según lo implementábamos. Primero, probamos que al escribirle cosas (sin que ejecutara ningún comando) no desapareciera el prompt. Una vez habíamos logrado esto, implementamos la opción para que ejecutara un mandato con N argumentos. Para probar esta funcionalidad, ejecutábamos mandatos del estilo (ls, find etc) en la MINISHELL y en la SHELL normal, para comprobar que coincidía el output. Posteriormente, probamos la redirección a entrada y salida estándar. Primero probamos la redirección por entrada estándar. Para testear esta funcionalidad usamos el mandato head. Después probamos salida estándar y error, para ello utilizamos el mandato cat, solo redirigiendo salida, solo redirigiendo error y redirigiendo ambas. Es decir, todas las combinaciones posibles.

Después, implementamos la funcionalidad de poder ejecutar 2 mandatos y enlazados por pipes, con la posibilidad de redirecciones de entrada y salida. Para probarlo, ejecutamos los comandos ls | wc y head | wc con todas las posibilidades de redirección de entrada y salida estándar que hemos mencionado antes. Los resultados obtenidos los íbamos comparando con los que obteníamos de la shell del sistema.

Para probar la funcionalidad de ejecutar más de 2 comandos enlazados por pipes y con redirecciones de entrada y salida. Usamos el mismo método que para probar la funcionalidad de dos mandatos, pero introduciendo un | wc al final.

Después implementamos el mandato cd. Para probarlo, usamos rutas absolutas y relativas. Además, al implementar un prompt que te mostraba la ruta actual, podíamos ir viendo rápidamente si la ruta se cambiaba o no. También usábamos ls para ver si el directorio de trabajo se había cambiado realmente.

Posteriormente, implementamos la funcionalidad de jobs y fg. Para probar el jobs, creábamos sleeps en segundo plano e íbamos viendo si se mostraban los que estaban en ejecución. También probamos a mandar al background mandatos con sleep y conectado por un pipe con un find. Para ver si el sleep tenía bloqueado o no al find. A la hora de probar el fg, lo que hemos hecho es ir pasando sleeps al fg a ver si se ejecutan bien. También probamos a mandar a fg mandatos que no estaban al final del array de procesos en segundo plano, con esto pudimos comprobar que cuando un mandato del medio del array era traído a primer plano, se eliminaba del array de jobs correctamente.

A la hora de probar la reprogramación de la señal producida por el ctrl + c, lo primero que hicimos era producir la señal en la minishell para ver si se cerraba. Después, probamos si terminaba correctamente los procesos en primer plano. Finalmente, comprobamos que no terminaba aquellos que estaban en backgroud. Para esto íbamos mirando el jobs y viendo si esos procesos seguían ejecutándose.

A la hora de probar el mandato exit, lo primero que hicimos era ver si se cerraba la minishell. Después, a la hora de hacer el exit, debugueamos el programa para ver si mataba a todos los hijos correctamente o si por el contrario se quedaban zombis.

El último mandato que implementamos fue el umask. Para probar este mandato, fuimos comparando los permisos de archivos creadas con la shell del sistema y la minishell con distintas máscaras. Para estas pruebas nos resultó muy útil el mandato ls -la y el mandato touch para ir creando los archivos.

Comentarios Personales

Por lo general, la realización de la práctica ha sido muy satisfactoria para ambos. Nos ha ayudado a entender como funcionan los pipes, y sobre todo cómo funciona la Shell del sistema. Tenemos que destacar que este tipo de prácticas siempre nos motivan, ya que construyes un sistema real que se puede usar y probar, cosa que no pasa en muchas asignaturas. Un aspecto que nos gustaría comentar es que nos han resultado liosos los comentarios de que esta práctica era más breve. Para nosotros en ningún momento lo ha sido, por lo que siempre hemos tenido la presión de si estábamos realizando bien las funciones ya que teníamos muchas más líneas de código que en la práctica anterior.

Entre los 2 hemos dedicado 44 horas a la realización de la práctica. Este tiempo incluye la aproximación de soluciones, la codificación del programa, las pruebas realizadas, los comentarios de código, las refactorizaciones y la realización de la memoria. También tenemos que destacar que al trabajar por parejas siempre hay un poco de desperdicio de tiempo, ya que hay que estar en comunicación constante.