Universidad
Rey Juan Carlos

ETSII: IC

SISTEMAS OPERATIVOS: MEMORIA PRÁCTICA 2

YAGO NAVARRETE MARTÍNEZ

SANTIAGO RAMOS GÓMEZ

ÍNDICE

AUTORES

Yago Navarrete Martínez:

Segundo año de Diseño y Desarrollo de Videojuegos + Ingeniería de Computadores.

Santiago Ramos Gómez:

Segundo año de Ingeniería de Computadores.

DESCRIPCIÓN DEL CÓDIGO

Al comienzo del código se incluyen los distintos **“#include”** necesarios para todas las funciones usadas, así como los distintos tipos de datos empleados a lo largo de la práctica. Las únicas variables globales empleadas son **“line”**, del tipo de datos **“tline”**, que contiene la información analizada de la línea de mandatos introducida tras el prompt por el usuario y **“WhereAmI”**, usada por el método **“redirect_cd()”**.

Las dos únicas dos funciones presentes en el código son **“redirect_cd()”**, **“kill_child()”** y **“main()”**. La primera se encarga de realizar el mandato interno de la Shell **“cd”**, la segunda es usada para el tratamiento de señales y la tercera contiene el código necesario para ejecutar los distintos mandatos introducidos por el usuario, realizando las redirecciones de entrada/salida pertinentes y conectándolos mediante pipes.

El código que presenta la función **“redirect_cd()”** usa como base el empleado para resolver el ejercicio **“mycd”** del tema tres, aunque modificamos la sintaxis empleada para acceder a los parámetros introducidos por el usuario. En dicho ejercicio, la dirección a la que realizar el cambio de directorio se encontraba en los argumentos pasados como parámetro al archivo. En el caso de la minishell, los parámetros se encuentran analizados en distintos campos de la variable **“commands”**, contenida en la variable **“line”**. Concretamente en los campos **“argc”** y **“argv”** similares a los de la función **“main”** pero referidos a cada mandato. El código comprueba que el número de argumentos sea correcto, que la dirección sea válida y busca el directorio **“HOME”** en caso de no recibir ningún parámetro. Por último, con **“chdir()”** modifica el directorio actual, avisa en caso de error e imprime por pantalla la dirección actual.

El archivo comienza a ejecutarse por la función **“main()”**, que se ha realizado empleando como base el código **“test.c”** proporcionado por los profesores. Se crean las variables necesarias para la ejecución de los mandatos, se guarda la entrada y salida estándar para recuperarlas en caso de redirección y se hace uso de **“signal()”** para evitar que la minishell muera al introducir las señales por teclado **“SIGINT”** y “**SIGQUIT”**. El flujo del programa a través del código sigue la estructura propuesta en el archivo proporcionado como base. Se emplea la función **“tokenize()”** para analizar la línea de comandos analizada guardada en un buffer y proporcionada por el usuario mediante teclado. Se realizan las redirecciones pertinentes si los campos analizados de la variable **“line”** no están vacíos. En caso de introducir una línea de mandatos a ejecutar en background se termina la ejecución del programa, pues la funcionalidad no ha sido implementada.

A continuación, se analiza el número de mandatos pasados como argumento. En caso de ser solo uno se comprueba si dicho comando es **“cd”** o **“exit”**, comparándolos directamente con dichas cadenas de caracteres. En caso de correspondencia, se realiza una llamada a la función **“redirect_cd()”** o se realiza un **“exit()”** para terminar la ejecución de la minishell, pues no se puede llevar a cabo mediante señales por teclado.

A la hora de ejecutar un único mandato se ha utilizado el código del ejercicio **"ejecuta"** del tema cuatro, en que se realiza un fork y se guarda el parámetro devuelto por la función, posteriormente se analiza si se ha llevado a cabo correctamente, se valida que el comando exista y se ejecuta mediante un **"execvp()"**.

Se tratan las señales indicadas en el enunciado, para que estas en lugar de ser ignoradas como ocurría mientras se esperaba la entrada del usuario, terminen de ejecutar el mandato que se esté llevando a cabo. Para ello al ser introducidas se redirigen a la función **"kill_child()"**. Y se devuelve el control de la Shell al usuario.

En caso de introducir más de un mandato en la línea de comandos, la ejecución de los mismos no es tan sencilla. Dado que se puede conectar la salida de cada uno con la entrada del inmediatamente siguiente, las redirecciones se han llevado a cabo mediante pipes. Para ello se ha asignado memoria dinámica a un array de punteros a punteros de enteros e iterado sobre el mismo para crear el número de pipes necesario en función de la cantidad de mandatos introducidos. Cada componente del array tendrá asociado un puntero a un espacio de memoria de dos enteros.

A continuación, un bucle comprueba que cada mandato sea válido y ejecuta un fork para crear un proceso hijo que ejecute dicho comando. En caso de que el proceso haya ocurrido sin problemas, se comprueba de que hijo se trata y en función de eso se abre un número de pipes distinto antes de ejecutar el mandato. Esto es porque el primer mandato solo necesita conectar su salida con el siguiente, el último solo necesita conectar su entrada, y los hijos intermedios tanto entrada como salida.

En primer lugar, se cierran las entradas y salidas de las tuberías que no van a llegar a usarse o que ya se han utilizado. Tras esto, se redirige la entrada o salida estándar a la tubería correspondiente. Se podría evitar el uso de **"stdin"** utilizando un 0 como valor en la función **"dup2()"** y en vez de **"stdout"** usar un 1. Esto es debido a que dichos valores enteros son los que toman por defecto los descriptores de fichero estándar. Tras las redirecciones pertinentes se ejecuta cada comando por separado, y se espera a que todos los hijos hayan terminado mediante un bucle while.

Se libera la memoria dinámica reservada y se limpian todas las salidas y entradas. Por último, estas se devuelven a sus valores por defecto en caso de haber sido modificadas. Todo el código se encuentra en bucle que espera la entrada de líneas de mandatos por parte del usuario, por lo que se vuelve a escribir el prompt y se vuelve al inicio del mismo.

COMENTARIOS PERSONALES

Problemas encontrados:

El apartado de realizar distintos mandatos de manera simultánea y conectarlos mediante pipes es el más complicado sin dudas. La dificultad para debuguear el código únicamente con la Shell no ayuda. Durante la elaboración del mismo empleamos multitud de funciones fprintf para saber cual era el flujo del mismo.

Crítica constructiva:

El reparto de puntos en la practica nos ha parecido muy adecuado. Unos cuantos apartados de esta estaban prácticamente resueltos solo habiendo atendido a la resolución de los ejercicios en clase de laboratorio. A pesar de esto, hemos realizado modificaciones sobre los mismos, especialmente sobre el mandato interno cd, para que el resultado final fuese más profesional. El trabajo con procesos hijos y pipes se dejaba al alumno, aunque gracias al archivo test.c sabías como enfocar la práctica desde un primer momento.

Propuesta de mejoras:

No tenemos propuestas reales de mejora para esta práctica. Los problemas de tiempo con los que nos hemos encontrado son resultado de una situación completamente ajena a los profesores de la asignatura, y por ello no creemos honesto pedir más tiempo de cara a alumnos en futuros años.

Evaluación del tiempo empleado:

La realización de la práctica parece muy amenazante de primeras y da la sensación de que va a requerir mucho tiempo. No obstante, este no es el caso en absoluto. Debido a una gran cantidad de prácticas y exámenes por la situación en que nos encontramos durante este cuatrimestre, no hemos conseguido sacar el tiempo necesario para implementar tareas en background. La elaboración del resto de apartados no requiere más de una semana de trabajo dosificado, pues gran cantidad del código a utilizar es prácticamente idéntico a ejercicios resueltos y entendidos en clases de laboratorio.