Tarea-Práctica 7

Sistemas Operativos

Shell de UNIX y función historial

1. Descripción

Este proyecto consiste en modificar un programa en **C** utilizado como interfaz *Shell* que acepta comandos de usuario y luego ejecuta cada comando como un proceso diferente. Una interfaz Shell proporciona al usuario un indicativo de comandos en el que el usuario puede introducir los comandos que desea ejecutar. El siguiente ejemplo muestra el indicativo de comandos $sh >$ En el que se ha introducido el comando de usuario $cat\ prog.c$. Este comando muestra el archivo en la terminal usando el comando cat de UNIX.

Una técnica para implementar una interfaz Shell consiste en que primero el proceso padre lea lo que el usuario escribe en la línea de comandos (por ejemplo: $cat\ prog.c$) y luego cree un proceso hijo separado que ejecute el comando. A menos que se indique lo contrario el proceso padre espera a que el hijo termine antes de continuar. Sin embargo normalmente las Shell de UNIX también permiten que el proceso hijo se ejecute en segundo plano concurrentemente especificando el símbolo $\&$ al final del comando reescribiendo el código anterior como:

$sh >\ cat\ prog.c\ \&$

Los procesos padre e hijos ejecutan de forma concurrente.

El proceso hijo se crea usando la llamada al sistema $fork()$ el comando de usuarios ejecuta utilizando una de las llamadas al usuario de la familia es $exec()$.

Se incluye un programa en sé que proporciona las operaciones básicas de un Shell de línea de comandos. este programa se compone de dos funciones: $main()$ y $setup()$ la función $setup()$ lee el siguiente comando del usuario (***que puede constar de hasta 80 caracteres***) lo analiza sintácticamente y lo descompone en identificadores separados que se usan para rellenar el vector de argumentos para el comando que se va a ejecutar. Si el comando se va a ejecutar en segundo plano terminará con un $\&$ y $setup()$ actualiza el parámetro $background$ de modo que la función $main()$ pueda operar de acuerdo con ello. Este programa se termina cuando el usuario introduce $< Control >< D > setup()$ invoca como consecuencia $exit()$

La función $main()$ presenta el indicativo de comandos $COMMAND- >$ y luego invoca $setup()$ que espera a que el usuario escriba un comando. Los contenidos del comando introducido por el usuario se cargan en la matriz $args$. Por ejemplo si el usuario escribe $ls\ -l$ el indicativo $COMMAND- >$ se asigna a $args[0]$ la cadena ls y se asigna a $args[1]$ y el valor $-l$; por cadena queremos decir una variable de cadena de caracteres de estilos **C** con carácter determinación nulo.

Este proyecto está organizado en dos partes 1) crear el proceso hijo y ejecutar el comando en este proceso y 2) modificar la shell para disponer de una función historial.

1.1. Creación de un proceso hijo

La primera parte de este proyecto consiste en modificar la función $main()$ indicada en el código de manera que al volver de la función $setup()$, se genera un proceso hijo y ejecuta el comando especificado

por el usuario.

Como hemos dicho anteriormente la función $setup()$ carga los contenidos de la matriz $args$ con el comando especificado por el usuario. Esta matriz $args$ se pasa a la función $execvp()$, que dispone de la siguiente interfaz:

$$execvp(char * command, char * params[\,]);$$

donde $command$ representa el comando que se va a ejecutar y para $params$ almacena los parámetros del comando. En este proyecto la función $execvp()$ debe invocarse con $execvp(args[0], args)$; hay que asegurarse de comprobar el valor de $background$ para determinar si el proceso padre debe esperar a que termine el proceso hijo o no.

1.2. Creación de una función historial

La siguiente tarea consiste en modificar el programa $shell.c$ para que proporcione una función *historial* que permita al usuario acceder a los, como máximo, 10 últimos comando que haya introducido. Estos comandos se numerarán comenzando por 1 y se incrementan hasta sobrepasar incluso 10; por ejemplo, si el usuario ha introducido 35 comandos, los últimos 10 comandos serán los enumerados desde el 26 hasta 35. Esta función historial se implementará utilizando unas cuantas técnicas diferentes.

En primer lugar, el usuario podrá obtener una lista de estos comandos cuando pulse $< Control >< C >$, que es la señal $SIGINT$. Los sistemas UNIX emplean **señales** para notificar a un proceso que se ha producido un determinado suceso. Las señales pueden ser síncronas o asíncronas, dependiendo del origen y de la razón por la que ya han señalado el suceso. Una vez que se ha generado una señal debido a que ha ocurrido un determinado suceso (*por ejemplo: una división por cero, un acceso a memoria ilegal, una entrada* $< Control >< C >$ del usuario, etcétera) la señal se suministra a un proceso donde será **tratada**. El proceso que recibe una señal puede tratarla mediante una de las siguientes técnicas:

- ignorar la señal,
- Usar la rutina de tratamiento de la señal predeterminada. o
- proporcionar una función específica del tratamiento de la señal.

Las señales pueden tratarse configurando en primer lugar determinados campos del la estructura:

$$C\ struct\ sigaction$$

y pasando luego esa estructura a la función $sigaction()$. Las señales se definen en el archivo:

$$/usr/include/sys/signal.h$$

Por ejemplo, la señal $SIGINT$ representa la señal para terminar un programa con la secuencia de control $< Control >< C >$. Una rutina predeterminada de tratamiento de señal para $SIGINT$ consiste en terminar el programa.

Alternativamente, un programa puede definir su propia función de tratamiento de la señal configurando el campo $sa_handler$ de $struct\ sigaction$ con el nombre de función que tratara la señal y luego invocando a la función $sigaction()$, pasándola 1) la señal para la que esté definida la rutina de tratamiento y 2) un puntero a $struct\ sigaction$.

Adjunto encontrarás un programa en C que usa la función $handle\_SIGINT$ para tratar la señal $SIGINT$. esta función escribe el mensaje çapturando control C", y luego invoca la función $exit()$ para terminar el

programa. Debemos usar la función $write()$ para escribir la salida en un lugar del habitual $printf()$, ya que la primera es segura con respecto a las señales, lo que quiere decir que se le puede llamar dentro de una función de tratamiento a una señal; $printf()$, no ofrece dichas garantías. Este programa ejecutar el bucle $while(1)$ hasta que el usuario introduzca la secuencia $< Control >< C >$. Cuando esto ocurre se invoca la función de tratamiento de señal $handle\_SIGINT$.

La función de tratamiento de señal se debe declarar antes de $main()$ y puesto que el control puede ser transferido a esta función en cualquier momento no puede pasarse ningún parámetro esta función. Por tanto, cualquier dato del programa que tenga que acceder, deberá declararse globalmente, es decir, al principio del archivo fuente, Antes de la declaración de función. Antes del volver de la función de tratamiento de señal, debe ejecutarse de nuevo el indicativo comandos.

Si el usuario introduces $< Control >< C >$, la rutina de tratamiento de señal proporcionará la lista de los últimos 10 comandos. Con esta lista el usuario puede ejecutar cualquiera de los 10 comandos anteriores escribiendo r donde x es la primera letra de dicho comando. Si hay más de un comando que comienza con x ejecuta el más reciente. También el usuario debe poder ejecutar de nuevo el comando más reciente escribiendo simplemente r. Podemos asumir que la r estará separada de la primera letra por sólo un espacio y que la letra irá seguida de $\backslash n$. Así mismo, si se desea ejecutar el comando más reciente, r irá inmediatamente seguida del $\backslash n$.

Cualquier comando que se ejecute de esta forma deberá enviarse como *eco* a la pantalla del usuario y el comando deberá incluirse en el búfer del historial como comando más reciente ($r\ x$ nos incluye en el historial, lo que se incluye es el comando al que realmente representa).

Si el usuario intenta utilizar está función historial para ejecutar un comando y se detecta que este es *erróneo*, debe proporcionarse un mensaje de error al usuario y no añadirse a la lista de historial y no debe llamarse a la función $excecvp()$. Estaría bien poder detectar los comandos incorrectamente definidos que se entreguen a $excecvp()$ que parecen válidos si no lo son, y no incluirlos en el historial tampoco, pero queda fuera de las capacidades de este programa). También debe modificarse $setup()$ de modo que devuelve un entero que indique si se ha creado con éxito una lista de $args$ válida o no y, la función principal debe actualizarse de acuerdo con ello.

2. Lineamientos para la entrega:

1. Se implementar lo solicitado