Trabajo práctico: desarrollo de un shell en c++

Sofia C. Arancibia
Facultad de Ingeniería y Ciencias Hídricas, *sofi.aran_05@hotmail.com*

***Resumen*—Este documento es una especificación del diseño y principales características de una shell desarrollada en c++.**
***Palabras clave*—fork, exec, pipe, comando, instrucción, prompt, proceso, librería.**

I. INTRODUCCIÓN

La aplicación presentada consiste en un intérprete de comandos estilo shell de Linux. Consiste en un bloque principal que contiene el bucle que se ejecutará hasta que el ususario ingrese el comando "exit" y finalice la aplicación. Tal como lo hace la shell de linux, esta aplicación imprime un prompt ("%") en cada ciclo del bucle principal indicando que está a la espera de una instrucción. La estructura modular de la aplicación facilita la lectura del código de la misma, ya que cada módulo ejecuta una única tarea. Se utilizan las librerías wait, stdio, unistd, cstdio, cstdlib, cerrno entre otras, para utilizar funciones de gestión de errores, ejecución de comandos (execvp), tuberías (pipes), creación de subprocesos (fork), etc.

II. FUENTES

Para facilitar la lectura, estructurado y depuración del proyecto, se utilizan funciones con objetivos específicos y únicos dentro de la aplicación.

A. Principal(main)

Como se mencionó anteriormente, aquí se ejecuta el bucle principal. Este bucle sólo finaliza cuando el usuario ingresa el comando "exit". En cada ciclo del bucle, se imprime el prompt y la aplicación queda a la espera de una intrucción. El usuario ingresa la instrucción que desea ejecutar como un string, para independizar al programa de la longitud de la instrucción. Éste string se convierte en cadena de caracteres para luego realizar el análisis necesario para la ejecución del comando. Utilizando la función *s_inst()*, se crea un arreglo de palabras, cuyo primer elemento se tomará como comando y los subsiguientes como argmentos del mismo. Si el comando no es "exit" el siguiente paso será ejecutar la instrucción, de lo cual se encargará el método *ejecutar()*. En el bucle principal se consideran además, casos especiales a considerar:

- Ingreso de comandos conectados mediante pipes: Para determinar esta situación se utiliza una rutina especial (*f_pipe()*) que modifica una variable booleana (*pi_pe*) en caso de encontrar el carácter '|' en la instrucción. Un valor *true* de la variable obligará a la aplicación a elaborar dos instrucciones independientes y conectarlas luego mediante una llamada a la rutina *tuberia()*.
- Ingreso del comando *exit:* La aplicación se cerrará.
- Ingreso del comando *cd:* Este comando no puede ejecutarse medianta la familia *exec()* por lo cual se utilizará la ruta indicada, como argumento de la rutina *chdir()* , la cual realiza la misma tarea que el comando *cd.*
- Ingreso del comando *cat >*: Para este caso particular, se crea un archivo cuyo nombre y extensión serán los indicados en la instrucción. Se realiza una lectura de contenido hasta el momento en que el usuario teclee [enter]; en ese momento el archivo se guardará en la ubicación en que el usuario esté en ese momento (es decir, en la carpeta en donde esté ejecutando el comando).
- Ingreso del comando *camino*: Este comando es propio de la aplicación. Se encarga de agregar o quitar una ruta, indicada por teclado, a la variable de entorno PATH. Su modo de uso se encuentra detallado en el manual de usuario que se encuentra en el paquete de la aplicación.

B. s_inst

El prototipo de esta función es:

*void s_inst(char *inst,char *ar[])*

La función no devuelve nada, por lo cual se ha declarado como *void*, y sus parámetros de entrada son una instrucción y una variable mediante la cual se devolverá el resultado calculado internamente.

El objetivo de este módulo es dividir la instrucción en palabras. Cada una de estas palabras será un término de la instrucción *inst.* Para la división se utiliza la función *char * strtok (char * str, const char * delimiters)* de la librería *cstrings:*

Una secuencia de llamadas a esta función divide 'str' en tokens, que son secuencias de caracteres contiguos separados por cualquiera de los caracteres que forman parte de 'delimiters'. En una primera llamada, la función espera una C string como argumento para str, cuyo primer carácter es usado como ubicación de inicio para buscar tokens. En las llamadas subsiguientes, la función espera el puntero null (para str) y usa la posición en la que quedó justo después de encontrar el último token como nueva ubicación de inicio para la búsqueda[1].

[1] http://www.cplusplus.com/reference/cstring/strtok/

C. ejecutar

El prototipo de esta función es:

*void ejecutar(char **args)*

Esta fucnión, al igual que la anterior, se declara como void, ya que no tiene valor de retorno. Su único argumento es el arreglo de palabras calculado con la función *s_inst.* La instrucción completa se ejecuta con la función *int execvp(const char *file, char *const argv[])* de la librería *unistd.h*:

El primer argumento, por convenio, debe apuntar al nombre de fichero asociado con el fichero que se esté ejecutando. El vector de punteros debe ser terminado por un puntero NULL[2]*. Como todas las demás funciones exec, execvp remplaza al proceso actual con un nuevo proceso. Esto tiene el efecto de correr un nuevo programa con el ID de proceso del proceso "reemplazante"*[3].

Por esto último es que se utiliza la función *execvp* luego de una llamada a *fork()* de modo que se reemplace un proceso que no afecte al curso del programa principal. De lo contrario, al terminar de ejecutarse el comando con *execvp* finalizaría la aplicación, imposibilitando al usuario ingresar una nueva instrucción.

Es posible que el usuario ingrese una instrucción de manera errónea (comando o argumentos equivocados), ante lo cual la aplicación le informará del caso mediante la llamada *char * strerror (int errnum)* de la librería *cstring:*

Interpreta el valor de errnum, generando un string con el mensaje que describe el error[4].

El mensaje se muestra en la pantalla utilizando la función *printf();*

D. tubería

El prototipo de esta función es:

*void tuberia(char *arg1[], char *arg2[])*

Esta función simplemente se encarga de crear la tubería que conectará dos comandos, los cuales son pasados como argumento de la función. El pipe tiene dos extremos: uno de lectura y otro de escritura. El extremo de lectura se cierra y el primer proceso envía su salida al extremo de escritura del pipe. El segundo proceso toma como su entrada la salida del extremo de lectura del pipe, para lo cual, el extremo de escritura se cierra. Aquí tambien se utiliza la función *fork()* ya que se ejecutarán dos procesos paralelos y ninguno de ellos debe afectar al curso normal de la aplicación.

[2] http://manpages.ubuntu.com/manpages/karmic/es/man3/exec.3.html
[3] https://support.sas.com/documentation/onlinedoc/sasc/doc700/html/lr2/zid-7307.htm
[4] http://www.cplusplus.com/reference/cstring/strerror/