KISH

Construcción de un Linux Shell

[Práctica 3]

Orlando Alemán Ortiz

DSO
2005 / 2006

Samuel Díaz Cabrera

Licencia

Esta obra ha sido publicada bajo licencia "Reconocimiento-NoComercial-CompartirIgual 2.5 Spain" de Creative Commons, la cual implica que:

Usted es libre de:
- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador.

No comercial. No puede utilizar esta obra para fines comerciales.

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

Y además:
- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.

Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by-nc-sa/2.5/es/ o envie una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

© 2007. Orlando Alemán Ortiz y Samuel Díaz Cabrera

Las Palmas de Gran Canaria, España.
orlando@pi314.es
http://orlando.pi314.es

Índice de contenido

Licencia.....2
Introducción.....4
  Previos.....4
  Terminología.....4
Construcción del shell.....6
  Diseño de la solución.....6
    Árbol de ejecución.....6
    Ejecución de órdenes.....7
  La solución: Kish.....8
    Visión general.....8
    Código fuente.....9
Anexo.....10
  Código fuente de Kish.....10
Referencias bibliográficas.....21

Introducción

Previos

Este documento constituye la memoria de la tercera práctica de la asignatura Diseño de Sistemas Operativos.

En esta ocasión la tarea que se nos plantea es la de de construir un mini-shell utilizando las llamadas al sistema FORK, EXEC, WAIT y PIPE. La solución que hemos propuesto se denomina "Kish". "Kish", cuyo nombre proviene de un juego de palabras entre *KISS* (*Keep It Simple Stupid*) y *shell,* es un mini-shell escrito en lenguaje C con las siguientes características:

- Pipe múltiple
- Redirección de entrada ("<") y de salida (">")
- Modo tanda ("&")
- Comando "cd" para cambiar el directorio de trabajo
- Búsqueda automática de programas en los directorios habituales ("/bin", "/usr/bin" y el actual)
- Posibilidad de ejecutar varias órdenes en una misma línea, utilizando el separador ";"

Terminología

Algunos de los términos que utilizaremos en esta memoria y que son de necesario conocimiento son los siguientes:

consola/shell:
Parte fundamental de un sistema operativo que ordena la ejecución de mandatos obtenidos del usuario por medio de una interfaz alfanumérica. Es decir, un shell es el intérprete de órdenes que se establece entre el usuario y el kernel.

entrada/salida estandar:
Por defecto la entrada de datos estandar se establece en el teclado y la salida de datos estandar en la pantalla del monitor, esto lo podemos variar a traves de tuberias o redirecciones. Por ejemplo, podemos hacer que la entrada sea el raton y la salida la impresora.

modo tanda/background:
Segundo plano. Se habla de proceso en segundo plano cuando se ejecuta sin nuestra interactividad o lo pasamos a modo suspendido.

nucleo:
Parte principal de un sistema operativo, encargado del manejo de los dispositivos, la gestión de la memoria, del acceso a disco y en general de casi todas las operaciones del sistema que permanecen invisibles para nosotros.

orden:

Instrucción que el usuario le da al sistema y que invoca la ejecución de uno o de más programas (si se encuentran conectadas mediante una tubería). Las órdenes se separan entre sí por ";" o siendo escritas en múltiples líneas.

señales:

Las señales son eventos que se hacen llegar a un proceso en ejecución para su tratamiento por este. Las señales las podemos mandar nosotros u otros programas a otros programas. Tienen diferentes valores, y en función a esos valores el proceso que las recibe actua de una manera u otra.

tubería/pipe:

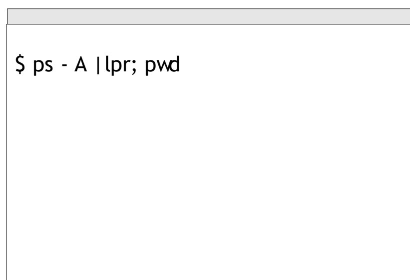
Las tuberías son como conexiones entre procesos. La salida de un proceso la encadenamos con la entrada de otro, con lo que podemos procesar unos datos en una sola linea de comando.

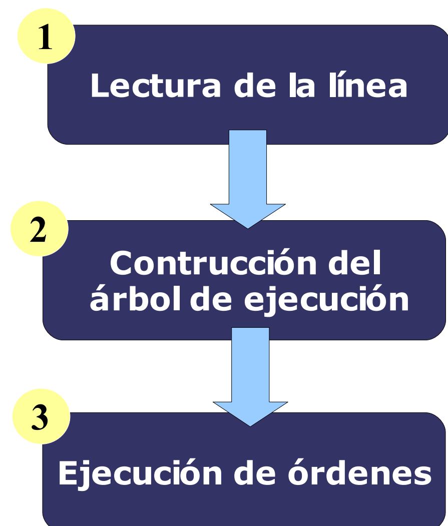
Construcción del shell

Diseño de la solución

El primer paso que debemos llevar a cabo es la descripción de la solución planteada. En nuestro caso, plantearemos el problema de la siguiente forma:

Un proceso principal (el que ejecuta el shell) estará continuamente leyendo las líneas que introduzcamos, entendiéndose por línea una o más órdenes separadas por punto y coma (“;”).Tras una lectura, se construirá un árbol de ejecución. Durante este proceso, se comprobará si la sintaxis que ha empleado el usuario es correcta y se obtendrá una estructura de representación más cómoda de las tareas a realizar por el shell: orden, dependencias, redireccionamientos, etc.





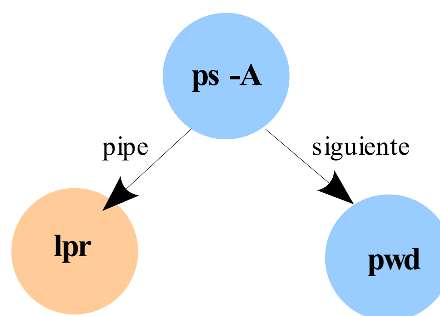
Durante la “ejecución de órdenes” se lanzan a ejecutar las órdenes a medida que se desciende por el árbol obtenido en la fase anterior, construyendo las rutas de los programas implicados y teniendo en cuenta las condiciones representadas en el árbol. Cada programa lanzado se ejecutará en un nuevo proceso.

Árbol de ejecución

Es básicamente una estructura de datos basada en un árbol binario de altura *n*, siendo *n* el número de órdenes a ejecutar. Cada nodo está formado por ocho campos, que son y significan lo siguiente:

info	El programa implicado
args[]	Argumentos que recibe
nargs	Número de argumentos
fentrada	Fichero del que recibe datos (redirección de entrada)
fsalida	Fichero al que envía datos (redirección de salida)
tanda	¿Se ejecuta en modo tanda?
hizq	Enlace hacia otro nodo. Entre el programa actual y el del nodo izquierdo existe un pipe (en sentido descendente)
hder	Enlace hacia otro nodo. Siguiente orden a ejecutar

Por ejemplo, la orden “ps -A | lpr; pwd” se derivaría en el árbol:



Ejecución de órdenes

Como se ha dicho, la ejecución de una línea se reduce a ejecutar el árbol de ejecución que genera siguiendo las directrices que el propio árbol establece:

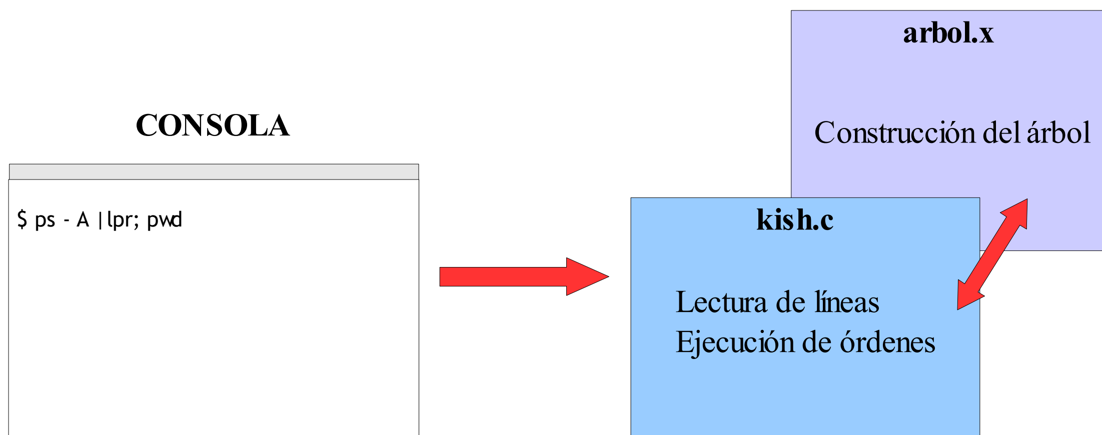
- Los nodos de la diagonal exterior derecha son los cabeza de comando
- Si un nodo tiene hijo izquierdo, la salida estándar del primero estará conectada con la entrada estándar del segundo.
- En caso de coincidir un pipe y una redirección en la misma sentencia, el primero tendrá mayor precedencia que el segundo.

Por la propia naturaleza de las tuberías, deberán crearse k-1 unidades para un pipe múltiple de k instrucciones. Y además, en cada nodo será necesario obtener la ruta del binario implicado.

La solución: Kish

Visión general

El código de Kish está distribuido en tres ficheros: kish.c, arbol.h y arbol.c. Los dos últimos forman evidentemente un módulo, así que podemos tratarlos como una unidad. Así pues, tenemos kish.c y arbol.x.



De **arbol.x** nos interesan básicamente la declaración de la estructura de datos que conforma los árboles y la operación de generación.

Un árbol no es más que una concatenación de structs construidos en memoria dinámica. Cada nodo tiene la forma:

```
struct Nodo {
  char *info;     // El comando
  char **args;    // Argumentos
  int nargs;      // Numero de argumentos
  char *fentrada; // En redirecciones, contiene el fichero de entrada
  char *fsalida;  // En redirecciones, contiene el fichero de salida
  bool tanda;     // Modo tanda (background)

  struct Nodo *hizq, *hder;
};
typedef struct Nodo NodoArbol;
```

Para construir y destruir un árbol de órdenes se dispone, respectivamente, de las funciones construyeArbol y destruyeÁrbol. El aspecto de estas operaciones es el siguiente:

```
NodoArbol* construyeArbol(char* linea);
void destruyeArbol(NodoArbol *raiz);
```

En **kish.c** debemos fijarnos en las funciones que muestra la siguiente tabla:

*void ejecutarArbol(NodoArbol *raiz, char **envp)*	
raiz: puntero a la nodo raiz del árbol **envp:** vector con las variables de entorno del shell	Realiza el descenso más general del árbol, el de la derecha. En cada nodo, invoca ejecutarNodo.
void ejecutarNodo(NodoArbol *nodo, char **envp)	
nodo: puntero a nodo orden **envp:** vector con las variables de entorno del shell	Realiza el procesado de la orden. Si es un comando simple (no pipe), invoca directamente a ejecutarComando. Si no, crea los pipes necesarios, crea tantos procesos hijo como programas implicados y lanza, ya en cada uno de ellos, ejecutarProceso.
void ejecutarProceso(int i, NodoArbol *nodointerno, char **envp)	
i: comando dentro del pipe, empezando por el 0. **nodointerno:** puntero al nodo cabeza de la orden que está siendo tratada. **envp:** vector con las variables de entorno del shell	Es básicamente el código que ejecutan los procesos hijos del shell implicados en un pipe.Prepara las redirecciones pertinentes y luego invoca a ejecutarComando.
void ejecutarComando(NodoArbol *nodo, char **envp)	
nodo: Nodo a ejecutar **envp:** vector con las variables de entorno del shell	Construye la ruta del programa, prepara redirecciones y crea un hijo para que ejecute la acción correspondiente.

Código fuente

[Ver anexo]

Anexo

Código fuente de Kish

kish.c

```
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <wait.h>

#include "arbol.h"

#define TAMLINEA 200


char * _path = "/bin/;/usr/bin/;";
int **vecpipes = NULL;
int *vecpids = NULL;
int npipes = 0;
int pipesalida[2];


char* construirRuta(char *cmd)
// INFO: Devuelve, si es posible, la ruta completa de un comando. Si no, devuelve NULL
{
    char *rutas, *p, *candidato=NULL;

    if((cmd[0]=='/')||((cmd[0]=='.')&&(cmd[1]=='/'))||
((cmd[0]=='.')&&(cmd[1]=='.')&&(cmd[2]=='/'))) {
        if(!access(cmd,X_OK)) {
            candidato = malloc((strlen(cmd)+1)*sizeof(char));
            strcpy(candidato, cmd);
        } else
            return NULL;
    } else {
        rutas = malloc((strlen(_path)+1)*sizeof(char));
        strcpy(rutas, _path);

        // Para cada ruta comprueba que exista el comando
        for(p = strtok(rutas, ";");  p;  p = strtok(NULL, ";")) {
            // Construimos un posible path completo para el comando
            candidato = malloc((strlen(p)+strlen(cmd)+1)*sizeof(char));
            strcpy(candidato, p);
            strcat(candidato, cmd);
```

```
            if(!access(candidato, X_OK)) {
                // Comprueba si el fichero candidato existe y es ejecutable
                // y en tal caso finaliza la bÃºsqueda del mismo
                break;
            }

            // Path no valido, liberamos memoria
            free(candidato);
            candidato = NULL;
        }
        free(rutas);
    }
    return candidato;
}



void ejecutarComando(NodoArbol *nodo, char **envp) {
    int status;
    char **args = NULL, *cmd;
    int pidhijo,fdin, fdout;

    if(strcmp(nodo->info, "exit")==0) {
        // Controlar parÃ¡metros de salida, matar hijos, etc
        exit(0);
    }

    if(strcmp(nodo->info,"cd")==0) {
        // Lo que haga el chdir
        if (!chdir(nodo->args[1])==0) {
            printf("SHELL: El PATH no existe\n");
        }
    } else {
        // Comando no pertenecen al Shell (Caso habitual)
        pidhijo = fork();
        assert(pidhijo!=-1);
        if (pidhijo!=0) {
            // Soy el padre
            if (!nodo->tanda) {
                // Espero por mi hijo
                waitpid(pidhijo, &status, 0);
            } else {
                // No espero por el hijo
                waitpid(pidhijo, &status, WNOHANG);
            }
        } else {
            //Soy el hijo
            cmd = construirRuta(nodo->info);
            if(cmd == NULL) {
                printf("SHELL: Comando desconocido\n");
                exit(1);
            }

            if(nodo->fentrada != NULL) {
                // Redireccion de entrada
                fdin = dup(0);
```

```
                close(0);
                if(open(nodo->fentrada, O_RDONLY, 0644) == -1) {
                    dup(fdin);
                    printf("SHELL: Error de acceso a fichero (%s)\n", nodo->fentrada);
                    exit(2);
                }
            }

            if(nodo->fsalida != NULL) {
                // Redireccion de salida
                fdout = dup(1);
                close(1);
                if(open(nodo->fsalida, O_CREAT|O_WRONLY, 0644) == -1) {
                    dup(fdout);
                    printf("SHELL: Error de acceso a fichero (%s)\n", nodo->fsalida);
                    exit(3);
                }
            }

            // Y finalmente ejecutamos el comando
            assert(execve(cmd, nodo->args, envp)==-1);
        }
        return;
    }
}



void inicializar(NodoArbol* nodo) {
    NodoArbol *aux = nodo;
    int i;

    // Eliminamos previos
    if(vecpipes!=NULL) {
        for(i=0; i<npipes; i++) {
            close(vecpipes[i][0]);
            close(vecpipes[i][1]);
            free(vecpipes[i]);
        }
        free(vecpipes);
        vecpipes = NULL;
    }

    // Contamos nÃºmero de pipes
    npipes = -1;
    while(aux!=NULL) {
        npipes++;
        aux = aux->hizq;
    }

    // Creamos nuevos pipes
    vecpipes = (int **) malloc(npipes*sizeof(int*));
    for(i=0; i<npipes; i++) {
        vecpipes[i] = (int *) malloc(2*sizeof(int));
        pipe(vecpipes[i]);
    }
```

```
    // Generacion del vector de PIDs para los hijos
    if(vecpids!= NULL)
        free(vecpids);
    vecpids = (int *) malloc((npipes+1)*sizeof(int));
}



void ejecutarProceso(int i, NodoArbol *nodointerno, char **envp) {
    int j;

    // PASO 1: Cerramos pipes que no son los que unen al proceso i con el i-1 y con
i+1
    for(j=0; j<i-1; j++) {
        close(vecpipes[j][0]);
        close(vecpipes[j][1]);
    }

    for(j=i+1; j<npipes; j++) {
        close(vecpipes[j][0]);
        close(vecpipes[j][1]);
    }

    // PASO 2: Fijamos el flujo de entrada y de salida que deseamos
    if(i!=0) {
        // Leo del pipe i-1
        dup2(vecpipes[i-1][0], 0);
        close(vecpipes[i-1][0]);
        close(vecpipes[i-1][1]);
    }

    if(i<npipes) {
        //Escribo en pipe i
        dup2(vecpipes[i][1], 1);
        close(vecpipes[i][1]);
        close(vecpipes[i][0]);
    }

    ejecutarComando(nodointerno, envp);
    // No es necesario cerrar pipes, pues el proceso finalizarÃ¡ a continuaciÃ³n
    return;
}



void ejecutarNodo(NodoArbol *nodo, char **envp) {
    int i, result;
    NodoArbol *nodoactual;

    if(nodo==NULL) {
        assert(0);
    }
    if(nodo->hizq==NULL) {
        ejecutarComando(nodo, envp);
        return;
```

```
    }

    // Necesitamos construir un vector de pipes (para comunicar procesos consecutivos)
    // y construir el vector vecpids
    inicializar(nodo);

    nodoactual = nodo;

    // Se crean los procesos que conforman el pipe
    for(i=0; i<npipes+1; i++) {
        vecpids[i] = fork();
        assert(vecpids[i]!=-1);
        if(vecpids[i]==0) {
            // CÃ³digo que ejecutarÃ¡ el hijo i
            ejecutarProceso(i, nodoactual, envp);
            exit(0);
        }
        nodoactual = nodoactual->hizq;
    }

    // Cerramos todos los pipes que no utilizamos
    for(i=0; i<npipes; i++) {
        close(vecpipes[i][0]);
        close(vecpipes[i][1]);
    }

    // Se espera por los hijos
    for(i=0; i<npipes+1; i++) {
        waitpid(vecpids[i], &result, 0);
    }
    return;
}



void ejecutarArbol(NodoArbol *raiz, char **envp) {
    if(raiz==NULL)
        return;
    ejecutarNodo(raiz, envp);
    ejecutarArbol(raiz->hder, envp);
    return;
}



int main(int argc, char **argv, char **envp) {
    char linea[TAMLINEA];
    NodoArbol *arbol = NULL;

    printf("\nBienvenido a Kish Shell\n\n");

    while(true) {
        // Presentacion del prompt
        // printf("%s@%s:~> ", getenv("LOGNAME"),getenv("HOST"));
        printf("%s@kish:~> ", getenv("LOGNAME"));
```

```
        // Lectura de una linea
        fgets(linea,TAMLINEA,stdin);
        linea[strlen(linea)-1]='\0';

        // Obtencion del arbol de descomposiciÃ³n
        arbol = construyeArbol(linea);

        // EjecuciÃ³n siguiendo el Ã¡rbol de descomposiciÃ³n
        ejecutarArbol(arbol, envp);

        // Eliminacion del arbol (obligatorio)
        destruyeArbol(arbol);
        fflush(NULL);
    }
    return 0;
}
```

arbol.h

```
#ifndef PARSER
#define PARSER

/* MAXARGS incluye el caracter NULL terminador y el primer argumento
   que es el propio ejecutable */
#define MAXARGS 20

#include <stdbool.h>


struct Nodo {
  char *info;     // El comando
  char **args;    // Argumentos
  int nargs;      // Numero de argumentos
  char *fentrada; // En redirecciones, contiene el fichero de entrada
  char *fsalida;  // En redirecciones, contiene el fichero de salida
  bool tanda;     // Modo tanda (background)

  struct Nodo *hizq, *hder;
};
typedef struct Nodo NodoArbol;


NodoArbol* construyeArbol(char* linea);
void destruyeArbol(NodoArbol *raiz);

#endif
```

arbol.c

```
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>
#include "arbol.h"
#include <stdio.h>

NodoArbol* construirNodo(char *info)
// INFO: Construye nodo para ejecutable
{
    NodoArbol *nuevo = (NodoArbol *) malloc(sizeof(NodoArbol));
    int i;

    nuevo->info = (char *) malloc((strlen(info)+1)*sizeof(char));
    strcpy(nuevo->info, info);

    nuevo->args = (char **) malloc(MAXARGS*sizeof(char*));
    nuevo->args[0] = (char *) malloc((strlen(info)+1)*sizeof(char));

    strcpy(nuevo->args[0], info);
    nuevo->nargs = 1;
    for(i=1; i<MAXARGS; i++) {
        nuevo->args[i] = NULL;
    }
    nuevo->tanda = false;
    nuevo->hder = NULL;
    nuevo->hizq = NULL;
    nuevo->fentrada = NULL;
    nuevo->fsalida = NULL;
    return nuevo;
}


NodoArbol* construyeArbol (char *linea)
// INFO: Construye un árbol de descomposición para la línea especificada
// POST: Requiere aplicar destruyeArbol cuando el árbol deje de ser usado
{
    NodoArbol *raiz = NULL;

    if(linea!=NULL) {
        char *buffer = (char *) malloc ((strlen(linea)+1)*sizeof(char));
        int linea_iter = 0, buffer_iter = 0;
        bool in_redir_gt = false, in_redir_lt = false, in_cmd = true, in_pipe = false,
comillas = false, bandera;
        NodoArbol *nuevo = NULL, *previo = NULL, *actual = NULL, *pipenuevo = NULL,
*pipeprevio = NULL;

        while(linea_iter < strlen(linea)+1) {
            bandera = false;

            // Tratamiento especializado (según el tipo de caracter)
            switch(linea[linea_iter]) {
            case '"':
                bandera = comillas;
                comillas = !comillas;
                break;
```

```
case '&':
    if(!comillas) {
        assert(raiz!=NULL);
        nuevo->tanda = true;
        break;
    }

case '>':
    if(!comillas) {
        assert(!in_redir_gt);
        if(buffer_iter > 0) {
            bandera = true;
        }
        break;
    }

case '<':
    if(!comillas) {
        assert(!in_redir_lt);
        if(buffer_iter > 0) {
            bandera = true;
        }
        break;
    }
case '|':
    if(!comillas) {
        if(in_cmd) {
            assert(buffer_iter>0);
            bandera = true;
        }
        break;
    }


case '\0':
case ' ':
case ';':
    if(!comillas) {
        bandera = true;
        break;
    }

default:
    buffer[buffer_iter] = linea[linea_iter];
    buffer_iter++;
}


// La bandera activa la realización de cambios en el árbol
if(bandera) {
    if(buffer_iter > 0) {
        buffer[buffer_iter] = '\0';
        if(in_cmd) {
            assert(! (in_redir_gt||in_redir_lt));

            if(in_pipe) {
```

```
                        in_cmd = false;
                        actual = construirNodo(buffer);
                        pipeprevio->hizq = actual;
                        pipeprevio = actual;
                    } else {
                        in_cmd = false;
                        nuevo = construirNodo(buffer);

                        if(previo == NULL) {
                            raiz = nuevo;
                        } else {
                            previo->hder = nuevo;
                        }
                        actual = previo = nuevo;
                    }

                } else {

                    if(in_redir_lt && actual->fentrada==NULL) {
                        actual->fentrada = (char *)
malloc((buffer_iter+1)*sizeof(char));
                        strcpy(actual->fentrada, buffer);
                    }
                    if(in_redir_gt && actual->fsalida==NULL) {
                        actual->fsalida = (char *)
malloc((buffer_iter+1)*sizeof(char));
                        strcpy(actual->fsalida, buffer);
                    }
                    if(!in_redir_gt && !in_redir_lt) {
                        actual->args[actual->nargs] = (char *)
malloc((buffer_iter+1)*sizeof(char));
                        strcpy(actual->args[actual->nargs], buffer);
                        actual->nargs++;
                    }
                }

                buffer_iter = 0;
            }

        }

        // Caracteres que requieren consideraciones posteriores
        switch(linea[linea_iter]) {
        case ';':
            in_cmd = true;
            in_redir_gt = false;
            in_redir_lt = false;
            in_pipe = false;
            break;
        case '>':
            if(!comillas) {
                in_redir_gt = true;
            }
            break;
        case '<':
            if(!comillas) {
```

```
                    in_redir_lt = true;
                }
                break;
            case '|':
                in_cmd = true;
                in_pipe = true;
                pipeprevio = actual;
                in_redir_lt = false;
                in_redir_gt = false;
                break;
            }
            linea_iter++;
        }
    }
    return raiz;
}


void destruyeArbol(NodoArbol *raiz)
// INFO: Destruye el árbol creado por la operación construyeArbol
{
    int i;

    if (!raiz)
        return;
    else {
        destruyeArbol(raiz->hizq);
        destruyeArbol(raiz->hder);
        free(raiz->info);
        free(raiz->fentrada);

        free(raiz->fsalida);
        for(i=0; i<raiz->nargs; i++) {
            free(raiz->args[i]);
        }

        free(raiz->args);
        free(raiz);
        raiz = NULL;
    }
}
```

Referencias bibliográficas

- **[*ComposLin*]** Glosario sobre Linux
  URL:http://www.escomposlinux.org/glosario

- **[*Wikipedia*]** "Wikipedia, la enciclopedia libre". Edición en español
  URL: http://es.wikipedia.org