

# La implementación de `smallsh`

20 de diciembre de 2007

## Índice

<b>1. Introducción</b>	<b>1</b>
<b>2. Funciones a implementar en la práctica</b>	<b>1</b>
2.1. <code>userfn.h</code> . . . . .	1
2.2. <code>userfn.c</code> . . . . .	3
2.3. <code>procline.c</code> . . . . .	4
2.4. <code>runcom.c</code> . . . . .	6
<b>3. Ficheros adicionales</b>	<b>7</b>
3.1. <code>smallsh.h</code> . . . . .	7
3.2. <code>smallsh.c</code> . . . . .	8
3.3. <code>shell.l</code> . . . . .	9
3.4. <code>gettok.h</code> . . . . .	9
3.5. <code>gettok.c</code> . . . . .	11
3.6. <code>buffer.h</code> . . . . .	16
3.7. <code>buffer.c</code> . . . . .	17
3.8. <code>Makefile</code> . . . . .	19

## 1. Introducción

Este documento describe el *shell* `smallsh` que se debe modificar para las prácticas de Sistemas Operativos. Este *shell* es capaz de leer líneas de entrada desde el teclado o desde un fichero y reconocer tanto las órdenes como los caracteres especiales (tuberías, redirección, etc.). Además, es capaz de ejecutar líneas de comandos sencillas (órdenes con sus argumentos) separadas por “;” o por “&”. En este último caso, las órdenes se ejecutarán en segundo plano.

Desde el punto de vista de la implementación, el *shell* se puede ver dividido en dos partes:

- una que implementa la lectura y el reconocimiento de la línea de órdenes y la ejecución de órdenes sencillas (ya implementada).
- otra que el alumno tendrá que modificar o completar.

Por lo general, los ficheros de la primera parte (listados en la sección 3) no necesitan ser modificados por parte del alumno, aunque éste puede decidir modificarlos según su criterio.

A continuación se exponen comentados los ficheros de los que consta `smallsh`. Se listan todos por completitud.

## 2. Funciones a implementar en la práctica

El código ya implementado del *shell* necesita que el alumno implemente una serie de funciones. Las definiciones de esas funciones se pueden encontrar en el fichero `userfn.h`.

### 2.1. userfn.h

---

```
#ifndef __USERFN_H
#define __USERFN_H

enum
{
    FLECHA_ARRIBA,
    FLECHA_ABAJO
};

/**
 * user_inicializar
 *
 * El shell llama a esta función al principio para que se realicen
 * las acciones de inicialización necesarias.
 */
void user_inicializar(void);

/**
 * user_finalizar
 *
 * El shell llama a esta función al final para que se realicen
 * las acciones de finalización necesarias.
 */
void user_finalizar(void);

/**
 * user_getPrompt
 *
 * Devuelve una cadena con el prompt a mostrar. La cadena la debe reservar
 * esta función con malloc.
 *
 * @return cadena de prompt
 */
char * user_getPrompt(void);

/**
 * user_flecha
 *
 * Devuelve la cadena que se debe mostrar en la línea de órdenes al pulsar
 * la flecha arriba o la flecha abajo.
 *
 * @param direccion indica si es flecha arriba o flecha abajo
 * @param patron patrón a buscar en la historia (anterior o posterior, según
 * el valor de "dirección"
 *
 * @return cadena a poner en la línea de órdenes
 */
char * user_flecha(int direccion, char* patron);

/**
 * user_nueva_orden
 *
 * Esta función es llamada cada vez que el usuario pulsa INTRO, e informa
 * de la orden que ha escrito el usuario. Esta función debe copiar la cadena
 * y no modificar la cadena que se le pasa.
 */
```

```

* @param orden La orden que ha escrito el usuario
*
*/
void user_nueva_orden(char * orden);
60

/**
* user_tabulador
*
* Devuelve (si procede) la cadena a añadir al pulsar el tabulador. La
* función recibe en "número" si el tabulador se ha pulsado para una orden
* (un 1) o si se ha pulsado para un argumento (un 2). El argumento "numtab"
* especifica el número de veces que se ha pulsado el tabulador (1 ó 2).
* En la primera pulsación la función completará si sólo hay una opción. Si
* no, emitirá un pitido. En la segunda pulsación, si hay varias posibilidades,
* se listarán todas ellas.
*
* @param parte parte de la cadena a la que se aplica el tabulador
* @param numero número del argumento: 1->orden, 2->argumento
* @param numtab número de veces que se ha pulsado el tabulador (1 ó 2)
*
* @return Parte restante de la cadena completada o NULL si no se puede
* completar
*/
char * user_tabulador(char * parte, int numero, int numtab);
80

#endif

/* $Id: userfn.h 1399 2007-12-20 09:45:07Z pedroe $ */
85

```

---

He aquí la explicación de las funciones:

- `void user_inicializar(void)`. Esta función es llamada por el *shell* al inicio para que el alumno implemente las funciones de inicialización necesarias (por ejemplo, la inicialización de la historia de órdenes, etc.).
- `void user_finalizar(void)`. Equivalente a la anterior, esta función es llamada por el *shell* cuando éste va a terminar su ejecución. Esta función es útil para el alumno para implementar cualquier proceso de terminación de su implementación (por ejemplo, guardar la historia de órdenes).
- `char * user_getPrompt(void)`. Retorna una cadena con el *prompt* que mostrará el *shell*. Esta función debe reservar memoria para la cadena retornada utilizando `malloc()`.
- `char * user_flecha(int direccion, char * patron)`. Retorna la cadena con la línea de órdenes que se mostrará tras que el usuario haya pulsado la tecla indicada por *direccion* (que puede ser `FLECHA_ARRIBA` ó `FLECHA_ABAJO`, como se ve en el listado anterior). La variable *patron* guarda el patrón a buscar en la historia. El alumno es el encargado de manejar la historia de órdenes y de devolver la cadena adecuada. De nuevo, esta función debe reservar la memoria para la cadena retornada utilizando `malloc()`.
- `void user_nueva_orden(char * orden)`. Esta función es llamada cada vez que el usuario pulsa `<ENTER>`. La cadena *orden* guarda la orden que ha escrito el usuario. Esta función **no puede modificar el contenido de** *orden*.
- `char * user_tabulador(char * parte, int numero, int numtab)`. Retorna la parte faltante que se completará a partir de la cadena *parte*, ya que el usuario ha pulsado la tecla `<TAB>`. El parámetro *numero* especifica si el argumento *parte* se refiere a una orden o un argumento. El parámetro *numtab*, que puede ser 1 ó 2. Cuando *numtab* valga 1, esta función debe emitir un pitido si hay más de una posibilidad a la hora de completar. Si es un 2, debe mostrar una lista de las posibilidades si hay más de una. En todos los casos en los que haya más de una posibilidad, la función retornará `NULL`. En cualquier otro caso (es decir, cuando se pueda completar de una única forma la parte dada), la función retornará

la cadena faltante a partir de la parte dada como argumento (de nuevo, la cadena retornada se tiene que reservar con `malloc()`).

## 2.2. userfn.c

Este fichero contiene implementaciones vacías de esas funciones.

---

```
#include "smallsh.h"

void
user_inicializar(void)
{
    /* Vacío */
}

void
user_finalizar(void)
{
    /* Vacío */
}

char *
user_getPrompt(void)
{
    /* Implementación */
    return NULL;
}

char *
user_flecha(int direccion, char* patron)
{
    /* Implementación */
    return NULL;
}

void
user_nueva_orden(char * orden)
{
    /* Implementación */
}

char *
user_tabulador(char * parte, int numero, int numtab)
{
    /* Implementación */
    return NULL;
}

/* $Id: userfn.c 1399 2007-12-20 09:45:07Z pedro $ */
```

---

## 2.3. procline.c

Este fichero lee la estructura `TokBuf` y construye una línea de comandos a ejecutar. En la implementación dada esta función sólo es capaz de ejecutar líneas sencillas y órdenes en segundo plano. El alumno tendrá que implementar todo lo especificado en la práctica.

---

```
#include "smallsh.h"

/* El siguiente procedimiento procesa una línea de entrada. La ejecución
 * termina cuando se encuentra un final de línea ('\n'). El car"cter ';'
 * sirve para separar unas órdenes de otras y el car"cter '&' permite
 * una orden en background.
 */
```

---

```

*/
void
procline(struct TokBuf * tb) {
    char * arg[MAXARG + 1]; /* Array de palabras: orden + argumentos. */
    TokType toktype;      /* Tipo de token. */
    int narg;             /* Número de argumentos leídos para la orden. */
    int where;           /* ¿ El proceso se ejecutar"s en primer o segundo plano ?.*/
    int ntoken;          /* Tokens procesados */

    narg = 0;
    for (ntoken = 0; ntoken < tb->length; ntoken++) {
        switch(toktype = tb->tokens[ntoken].type) {
            case ARG: /* Se ha leído un argumento. */
                if (narg < MAXARG)
                    arg[narg++] = tb->tokens[ntoken].data;
                break;
            case EOL:
            case SEMICOLON:
            case AMPERSAND:

            case ARROBA:
            case AND:
            case OR:
                where = (toktype == AMPERSAND) ? BACKGROUND :
                    FOREGROUND;
                if (narg != 0) {
                    arg[narg] = NULL;
                    runcommand(arg, where);
                }
                /* Seguir con la siguiente orden. Esto se da
                 * si se han introducido varias órdenes
                 * separadas por ';' o '&'. */
                narg = 0;
                break;
            default:
                ; /* Ignorar */
        }
    }
}

/* $Id: procline.c 1399 2007-12-20 09:45:07Z pedroe $ */

```

Para entender cómo realiza su labor `procline()`, y cómo está construida la estructura `TokBuf`, a continuación se muestra su definición (se puede ver en el fichero `gettok.h`, página 9):

```

struct Token
{
    TokType    type;      /* Tipo del token */
    char       * data;    /* Texto del token */
};

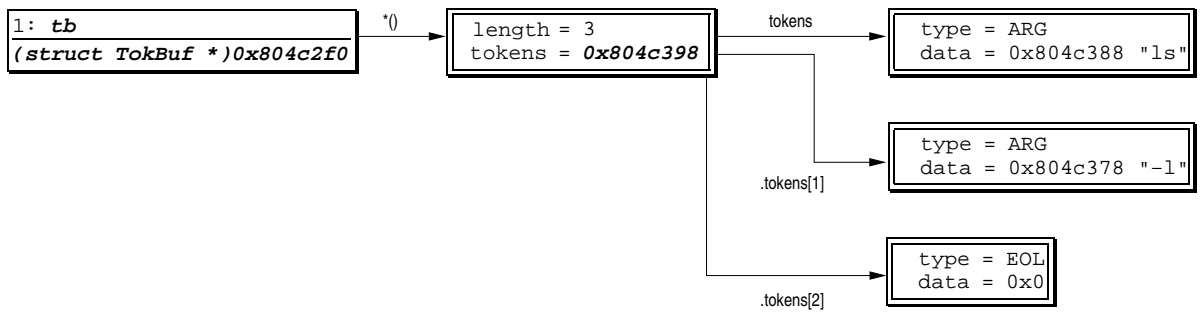
struct TokBuf
{
    int        length;    /* Longitud del array de tokens */
    struct Token * tokens; /* Array de tokens */
};

```

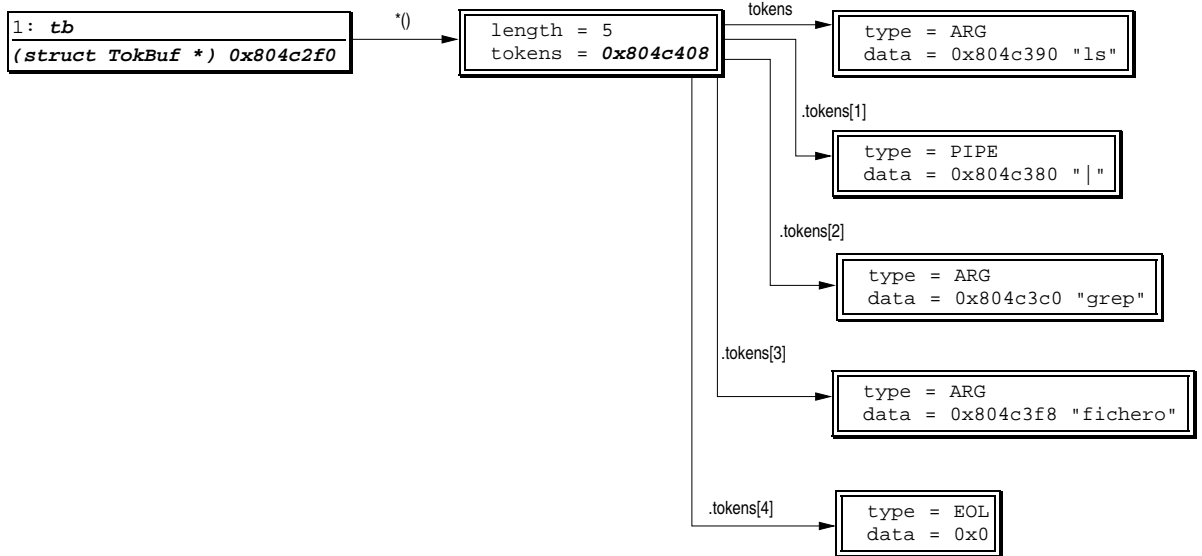
Como se ve, `TokBuf` guarda un array de *tokens*. Un *token* es un grupo de caracteres que tiene importancia para el *shell*, como por ejemplo, un argumento, un carácter de tubería, un punto y coma, etc. El código actual es capaz de leer la línea de entrada del usuario y construir esta estructura `TokBuf`.

A continuación se muestra cómo estaría configurada la estructura con diversas líneas de órdenes:

■ `ls -l`



■ `ls | grep fichero`



Nótese cómo el campo `type` contiene el identificador del token del que se trata. Finalmente, todas las listas de tokens terminan en el token nulo.

Aunque los identificadores se pueden ver en el fichero `gettok.h` en la página 9, se muestran a continuación junto con su explicación:

Identificador	Significado
EOL	especifica el final de la línea de comandos.
ARG	especifica un argumento o una órden.
AMPERSAND	el carácter “&”.
SEMICOLON	el carácter “;”.
PIPE	el carácter “ ”.
BACKQUOTE	el carácter “`”.
QUOTE	el carácter “'”.
MENOR	el carácter “<”.
MAYOR	el carácter “>”.
MAYORMAYOR	el conjunto de caracteres “>>”.
DOSMAYOR	el conjunto de caracteres “2>”.
DOSMAYORMAYOR	el conjunto de caracteres “2>>”.
AND	el conjunto de caracteres “&&”.
OR	el conjunto de caracteres “  ”.
ARROBA	el carácter “@” seguido del número de minutos.
PORCIENTO	el carácter “%”.
SPACE	caracteres de espacio (espacios, tabuladores, etc.).

## 2.4. runcom.c

La función `runcommand()` ejecuta una línea de comandos utilizando la función estándar `exec()`.

```
#include "smallsh.h"

/* Ejecuta una orden. "cline" es una array de palabras que contiene el nombre
 * de la orden y los parámetros de dicha orden. "where" indica si dicha
 * orden se debe ejecutar en primer o segundo plano.
 */
int
runcommand(char **cline, int where) {
    int pid, exitstat, ret;

    if ((pid = fork()) < 0) {
        perror("smallsh");
        return(-1);
    }

    if (pid == 0) {
        /* Estamos en el hijo. */
        execvp(*cline, cline);
        /* Ejecutamos la orden. */
        perror(*cline);
        /* Se llega aquí si no se ha podido
         * ejecutar. Por tanto, se ha producido
         * un error.*/
        exit(127);
    }

    /* Estamos en el padre. Si la orden se ejecuta en segundo plano, no
     * debemos esperar por lo que mostramos el pid del nuevo proceso y
     * regresamos. */
    if (where == BACKGROUND) {
        printf("[Identificador de proceso: %d]\n", pid);
        return(0);
    }

    /* Si la orden se ejecuta en primer plano, debemos esperar a que
     * termine.*/
    while ((ret = wait(&exitstat)) != pid && ret != -1)
        ;
}
```

```
    return(ret == -1 ? -1 : exitstat);
}
```

40

```
/* $Id: runcom.c 1399 2007-12-20 09:45:07Z pedroe $ */
```

---

### 3. Ficheros adicionales

Los ficheros que se listan a continuación son parte del *shell*, aunque el alumno no necesita modificarlos. Todas las mejoras propuestas en el enunciado de la práctica se pueden realizar modificando los ficheros de la sección anterior o añadiendo otros nuevos, sin necesidad de modificar los que se listan a continuación. Se incluyen por completitud y para que el alumno tenga una idea completa del funcionamiento interno del *shell*.

#### 3.1. smallsh.h

Contiene las definiciones comunes para todos los módulos *.c* del *shell*.

---

```
#ifndef __SMALLSH_H
#define __SMALLSH_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Definición del buffer de tokens */
#include "gettok.h"

/* Funciones del usuario */
#include "userfn.h"

/* Funciones del buffer */
#include "buffer.h"

#define MAXARG 512

#define FOREGROUND 0
#define BACKGROUND 1

void procline(struct TokBuf*);
int runcommand(char **, int);

#endif

/* $Id: smallsh.h 1398 2007-12-20 08:06:01Z pedroe $ */
```

---

#### 3.2. smallsh.c

Este fichero es el que implementa la función principal (`main()`) del programa. Sigue un bucle en el que, tras llamar a `userin()` para conseguir la línea de entrada por parte del usuario, llama a `gettok()` para convertir la cadena de caracteres escrita por el usuario en una estructura `TokBuf`. A continuación se llama a `procline()` con la estructura `TokBuf` retornada por `gettok()`. (La definición de `TokBuf` está en el fichero `gettok.h` en la sección 3.4 (pág. 9).

---

```
#include "smallsh.h"
```

```
int
```

```

main(int argc, char **argv)
{
    struct TokBuf *tb;
    struct Buffer *buf;

    /* Llamar a la función de inicialización del shell */
    user_inicializar();

    /* Establecer modo no interpretado en la entrada */
    modoInterpretado(0, 0);

    /* Procesar órdenes */
    while ((buf = userin()) != NULL)
    {
        tb = gettok(buf->data);

        procline(tb);

        /* Liberar el TokBuf que creó la llamada a userin() */
        liberaTokBuf(tb);
        free (tb);
        liberaBuffer(buf);
        free (buf);
    }

    /* Restaurar el modo de la entrada */
    modoInterpretado(0, 1);

    /* Finalmente, a la función de finalización */
    user_finalizar();

    /* Retornar una salida sin error */
    return 0;
}

/* $Id: smallsh.c 1398 2007-12-20 08:06:01Z pedro $ */

```

### 3.3. shell.l

Para diferenciar los argumentos de los caracteres especiales y para reconocer todos los caracteres de la línea de entrada se utiliza el programa `lex` (en concreto la implementación `flex`). El siguiente fichero se muestra por completitud:

```

%{
#include "gettok.h"
%}

%%

'      trataToken( QUOTE, yytext );
&      trataToken( AMPERSAND, yytext );
;      trataToken( SEMICOLON, yytext );
\|\|   trataToken( OR, yytext );
&&     trataToken( AND, yytext );
\|     trataToken( PIPE, yytext );
`      trataToken( BACKQUOTE, yytext );
\>     trataToken( MAYOR, yytext );
\<      trataToken( MENOR, yytext );
\>\>   trataToken( MAYORMAYOR, yytext );
2\>   trataToken( DOSMAYOR, yytext );
2\>\>  trataToken( DOSMAYORMAYOR, yytext );
@[0-9]+ trataToken( ARROBA, yytext );
\%     trataToken( PORCIENTO, yytext );

```

```
[ \t]+      trataToken( SPACE, yytext );
.          trataChar( yytext[ 0 ] );
```

```
%%
```

```
/* $Id: shell.l 1399 2007-12-20 09:45:07Z pedroe $ */
```

25

### 3.4. gettok.h

El fichero `gettok.h` contiene la definición de la estructura `TokBuf`, y operaciones que se conectan con el fichero anterior para ir la construyendo en base a la entrada del usuario.

```
#ifndef __TOKBUF_H
#define __TOKBUF_H
```

```
#include "buffer.h"
```

```
/* Tokens reconocidos */
```

```
enum TokType_
```

```
{
```

```
    EOL,          /* Fin de línea */
```

```
    ARG,          /* Argumento */
```

```
    AMPERSAND,   /* & */
```

```
    SEMICOLON,  /* ; */
```

```
    PIPE,        /* | */
```

```
    BACKQUOTE,  /* ` */
```

```
    QUOTE,       /* ' */
```

```
    MENOR,       /* < */
```

```
    MAYOR,       /* > */
```

```
    MAYORMAYOR, /* >> */
```

```
    DOSMAYOR,   /* 2> */
```

```
    DOSMAYORMAYOR, /* 2>> */
```

```
    AND,         /* && */
```

```
    OR,          /* || */
```

```
    ARROBA,     /* @ */
```

```
    PORCIENTO,  /* % */
```

```
    SPACE       /* " ", \t, etc. */
```

```
};
```

```
/* Tipo token */
```

```
typedef enum TokType_ TokType;
```

```
/* Estructura para almacenar un token junto con su contenido (si procede) */
```

```
struct Token
```

```
{
```

```
    TokType    type;          /* Tipo del token (ver arriba) */
```

```
    char       *data;        /* Texto del token */
```

```
};
```

```
/* Estructura que guarda un conjunto de tokens. Este conjunto representa
```

```
 * a la línea de comandos una vez que se ha hecho el parsing.
```

```
 */
```

```
struct TokBuf
```

```
{
```

```
    int        length;
```

```
    struct Token * tokens;
```

```
};
```

```
/**
```

```
 * userin
```

```
 *
```

```
 * Retorna el Buffer resultado de leer la cadena entrada por el teclado
```

```
 * Esta función llama a las funciones de userfn.h dependiendo de los
```

5

10

15

20

25

30

35

40

45

50

```

* caracteres leídos.
*
* @return El Buffer construido
*/
struct Buffer * userin();

/**
* trataToken
*
* Añade un token nuevo a curTokBuf. curTokBuf guarda el TokBuf actual
* siendo reconocido a partir de la entrada del usuario. Si hay una
* cadena temporal en tmpArg, la añade antes como un token de tipo ARG.
*
* @param type Tipo del token
* @param string Cadena del token
*/
void trataToken(TokType type, char *string);

/**
* trataChar
*
* Añade un car"cter al argumento temporal tmpArg.
*
* @param c El car"scter.
*/
void trataChar(char c);

/**
* gettok
*
* Retorna un TokBuf construido a partir de la cadena dada como par"smetro
*
* @param str La cadena
*
* @return El TokBuf* con la lista de tokens.
*/
struct TokBuf * gettok(char *str);

/**
* liberaTokBuf
*
* Libera la memoria asociada a un TokBuf
*
* @param t El TokBuf.
*/
void liberaTokBuf(struct TokBuf * t);

/**
* modoInterpretado
*
* Establece el modo (interpretado o no) en el descriptor de fichero "fd"
*
* @param fd El descriptor de fichero
* @param on 1=on, 0=off
*/
void modoInterpretado(int fd, int on);

#endif

/* $Id: gettok.h 1399 2007-12-20 09:45:07Z pedroe $ */

```

### 3.5. gettok.c

gettok.c implementa las funciones anteriores, incluyendo userin() y gettok().

---

```
#include "smallsh.h"

static char * blanco = "                                                                ";

/* Mantener contento al compilador */
extern int yy_scan_string();
extern int yylex();

static char *
parteTab(char * d, int c)
{
    char * q = d + c - 1;

    while (c-- && (*q != ' '))
        --q;
    return ++q;
}

static int
numeroTab(char * d, int c)
{
    char * q = d + c - 1;

    while (c--)
        if (*q-- == ' ')
        {
            /* ¿Espacios hasta el inicio? */
            while (c--)
                if (*q-- != ' ')
                    return 2;
            return 1;
        }

    return 1;
}

/* "userin" imprime un mensaje de petición de órdenes (prompt) y lee una línea.
 * Dicha línea es apuntada por ptr. */
struct Buffer *
userin()
{
    char c;
    int v, tabstatus, count, search, dir;
    char * prompt, * tmpstr;
    struct Buffer *buf;
    struct Buffer pattern;

    prompt = user_getPrompt();
    printf(" %s ", prompt);
    fflush(stdout);

    buf = creaBuffer (NULL);
    tabstatus = 0;
    count = 0;
    search = 0; /* Estado de búsqueda con las flechas */

    while (1 == (v = read(0, &c, 1)))
    {
        /* Flecha arriba: 27 91 65 */
        /* Flecha abajo: 27 91 66 */
        /* Backspace: 127 */
        /* Tabulador: 9 */

```

```

/* Identificar un car"acter normal o una pulsación de
 * tecla especial
 */
switch (c)
{
case 27:
    /* flechas */
    tabstatus = 0;
    tmpstr = 0;
    dir = -1;

    read(0, &c, 1); /* 91 */
    if (c != 91)
        break;
    read(0, &c, 1);
    if (c != 65 && c != 66)
        break;
    switch (c)
    {
    case 65: /* arriba */
        dir = FLECHA_ARRIBA;
        break;
    case 66: /* abajo */
        dir = FLECHA_ABAJO;
        break;
    }
    if (dir != -1)
    {
        if (!search)
        {
            /* Si no hay ninguna búsqueda */
            search = 1;
            iniciaBuffer (&pattern, buf->data);
        }
        tmpstr = user_flecha(dir, pattern.data);
    }

    liberaBuffer(buf);
    if (tmpstr)
    {
        count = strlen(tmpstr);
        iniciaBuffer(buf, tmpstr);
        free(tmpstr);

        printf("\r %s\r %s { %s } %s",
            blanco, prompt, pattern.data, buf->data);
    } else {
        count = 0;
        search = 0;
        iniciaBuffer (buf, NULL);
        liberaBuffer (&pattern);
        printf("\r %s\r %s %s", blanco, prompt, buf->data);
    }
    break;

case 9:
    /* TAB */
    {
        int i = numeroTab(buf->data, count);
        tmpstr = parteTab(buf->data, count);

        if (tabstatus != 2)
            tabstatus++;
        tmpstr = user_tabulador(tmpstr, i, tabstatus);
        if (tmpstr)

```

```

        {
            count += strlen(tmpstr);
            anyadeCadena(buf, tmpstr);
            tabstatus = 0;
            free(tmpstr);
        }
    }
    if (search)
    {
        search = 0;
        liberaBuffer (&pattern);
    }
    printf("\r %s\r %s %s", blanco, prompt, buf->data);
    break;
case 127:
    /* bs */
    tabstatus = 0;
    if (count)
    {
        --count;
        eliminaUltimo(buf);
        if (search)
            printf("\r %s\r %s %s",
                blanco, prompt, buf->data);
        else
            printf ("\b \b");
    }
    if (search)
    {
        search = 0;
        liberaBuffer (&pattern);
    }
    break;
case 10:
    /* Enter */
    tabstatus = 0;
    if (search)
    {
        search = 0;
        liberaBuffer (&pattern);
    }
    break;
default:
    tabstatus = 0;
    ++count;
    anyadeChar(buf, c);
    if (search)
        printf("\r %s\r %s %s",
            blanco, prompt, buf->data);
    else
        printf ("%c", c);
    if (search)
    {
        search = 0;
        liberaBuffer (&pattern);
    }
}

fflush(stdout);

/* Fin al pulsar el intro */
if (c == 10)

```

```

        {
            printf("\n");
            break;
        }
    }
}
195

/* Liberar el prompt. Se produce en cada ejecución de comando */
free(prompt);
200

if (v == -1)
    fprintf(stderr, "Error leyendo de la entrada");
205

if (v != 1)
{
    liberaBuffer(buf);
    return 0;
}
210

/* Informar de una nueva línea escrita */
user_nueva_orden(buf->data);
215

return buf;
}

/*
 * Buffer actual de tokens
 */
static struct TokBuf * curTokBuf;
220

/*
 * Buffer de argumento temporal
 */
static struct Buffer tmpArg;
225

/*
 * Añade un token nuevo a curTokBuf
 */
static void
anyadeToken(TokType type, char *string)
235
{
    int len;

    len = curTokBuf->length;
240

    /* Copiar el contenido del buffer en el token actual */
    curTokBuf->tokens =
        realloc(curTokBuf->tokens, (len + 1) * sizeof(struct Token));
    curTokBuf->length++;
245

    /* Crea el token len-ésimo */
    curTokBuf->tokens[len].type = type;
    if (string)
    {
        curTokBuf->tokens[len].data = (char *) malloc(strlen(string) + 1);
        strcpy(curTokBuf->tokens[len].data, string);
250
    } else {
        curTokBuf->tokens[len].data = 0;
    }
}
255

void
trataToken(TokType type, char *string)
{
    /* Limpiar el argumento temporal antes de añadir el token actual. */
260

```

```

    if (tmpArg.len != 0)
    {
        anyadeToken( ARG, tmpArg.data );

        liberaBuffer(&tmpArg);
        iniciaBuffer(&tmpArg, NULL);
    }

    /* Ignorar espacios */
    if ( type == SPACE )
        return;

    anyadeToken( type, string );
}

/*
 * Añade un caracter al buffer temporal de argumentos tmpArg
 */
void
trataChar(char c)
{
    anyadeChar(&tmpArg, c);
}

struct TokBuf *
gettok(char *cadena)
{
    /* Inicializar el buffer de tokens a retornar */
    curTokBuf = (struct TokBuf *)malloc(sizeof(struct TokBuf));
    curTokBuf->length = 0;
    curTokBuf->tokens = 0;

    /* Inicializar el argumento temporal a vacío */
    iniciaBuffer(&tmpArg, NULL);

    yy_scan_string (cadena);
    yylex ();

    /* Insertar un token EOL */
    trataToken(EOL, NULL);

    /* Liberar el buffer temporal */
    liberaBuffer( &tmpArg );

    return curTokBuf;
}

void
liberaTokBuf(struct TokBuf * t)
{
    int i;

    for (i = 0; i < t->length; i++)
        free(t->tokens[i].data);

    free(t->tokens);
}

#include <sys/ioctl.h>
#include <termio.h>

void
modoInterpretado(int fd, int on)
{

```

```

static struct termio parametros_ant;
static int status = 1;
struct termio parametros;
330

if ((on && status) || (!on && !status) || (!isatty(fd)))
    return;
if (on)
    ioctl(fd, TCSETA, &parametros_ant);
else {
335
    /* M'srquez, p'sg. 167 */
    ioctl(fd, TCGETA, &parametros_ant);
    parametros = parametros_ant;
    parametros.c_lflag &= ~(ICANON | ECHO);
    parametros.c_cc[4] = 1;
    ioctl(fd, TCSETA, &parametros);
340
}
status = on;
}
345

/* $Id: gettok.c 1399 2007-12-20 09:45:07Z pedro $ */

```

---

### 3.6. buffer.h

Los ficheros `buffer.h` y `buffer.c` implementan un tipo abstracto de datos “Buffer extensible”, en el sentido de que maneja automáticamente la memoria dinámica asignada al mismo: se le pueden añadir caracteres o cadenas y se reserva automáticamente espacio para las mismas. Así, se libera al cliente del tipo abstracto de datos de las cuestiones de manejo de memoria dinámica. El alumno puede añadir más operaciones conforme las necesite.

```

#ifndef __BUFFER_H
#define __BUFFER_H

struct Buffer
{
5
    int len;
    int max;
    char * data;
};
10

/* Crea un buffer vacío */
struct Buffer * creaBuffer(char* buf);

/* Inicia un buffer vacío */
void iniciaBuffer(struct Buffer* b, char* buf);
15

/* Elimina un buffer */
void liberaBuffer(struct Buffer * b);

/* Añade un carácter al final */
20
struct Buffer * anyadeChar(struct Buffer * b, char c);

/* Añade una cadena al final */
struct Buffer * anyadeCadena(struct Buffer * b, char * c);
25

/* Elimina el último carácter del final */
struct Buffer * eliminaUltimo(struct Buffer * b);

#endif
30

/* $Id: buffer.h 1399 2007-12-20 09:45:07Z pedro $ */

```

---

### 3.7. buffer.c

La única cuestión interesante de la implementación es la función `resize()`, que asegura que al menos quepan `b->len` bytes en el buffer. Esta función la utilizan como auxiliar todas las demás, para asegurar en todo momento que el tamaño del buffer es suficiente para albergar lo que se requiere de él.

---

```
#include "buffer.h"
#include <string.h>
#include <stdlib.h>

/* Tamaños de buffer hasta 1MB */
static int pot2[] = { 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192,
                    16384, 32768, 65536, 131072, 262144, 524288, 1048576 };

struct Buffer *
resize(struct Buffer * b)
{
    int i;

    if ((b->len + 1) < b->max)
        return b;

    i = 0;
    while (pot2[i++] <= (b->len + 1))
        ;

    i--;
    b->max = pot2[i];
    b->data = (char *)realloc(b->data, b->max);

    return b;
}

void
_iniciaBuffer (struct Buffer* b, char * str)
{
    b->len = 0;
    b->max = pot2[0];
    b->data = (char *)malloc(b->max);

    b->data[0] = 0;
    if (str != NULL)
        anyadeCadena(b, str);
}

/* Crea un buffer vacío */
struct Buffer*
creaBuffer (char* str)
{
    struct Buffer* tmp = (struct Buffer*)malloc (sizeof(struct Buffer));
    _iniciaBuffer (tmp, str);
    return tmp;
}

/* Inicia un buffer vacío */
void
iniciaBuffer(struct Buffer* b, char* str)
{
    _iniciaBuffer (b, str);
}

/* Libera el contenido de un buffer (no la estructura en sí) */
void
liberaBuffer(struct Buffer *b)
```

```

{
    free(b->data);
    b->len = 0;
    b->max = 0;
    b->data = NULL;
}
60
65

/* Añade un car"acter al final */
struct Buffer *
anyadeChar(struct Buffer *b, char c)
{
    b->len++;
    resize(b);
    b->data[b->len - 1] = c;
    b->data[b->len] = '\0';
    return b;
}
70
75
80

/* Añade una cadena al final */
struct Buffer *
anyadeCadena(struct Buffer* b, char* c)
{
    int len = strlen(c);
    b->len += len;
    resize(b);
    strcat(b->data, c);
    return b;
}
85
90
95

/* Elimina el último car"acter del final */
struct Buffer *
eliminaUltimo(struct Buffer * b)
{
    if (b->len > 0)
        b->data[--b->len] = '\0';
    return resize(b);
}
100
105

/* $Id: buffer.c 1398 2007-12-20 08:06:01Z pedro $ */

```

### 3.8. Makefile

Fichero para la utilidad `make`. Compila todo el código.

```

#!/usr/bin/make -f
# Fichero Makefile para construir el ejecutable smallsh

CFLAGS=-g -Wall
LDLIBS=-lfl # Para flex
OBJECTS=smallsh.o gettok.o procline.o runcom.o buffer.o userfn.o shell.o
smallsh: $(OBJECTS)

```

```
gettok.o: gettok.c
procline.o: procline.c
runcom.o: runcom.c 15
smallsh.o: smallsh.c
buffer.o: buffer.c 20
userfn.o: userfn.c
shell.c: shell.l
clean: 25
    rm -rf ** $(OBJECTS) shell.c smallsh core
# $Id: Makefile 1398 2007-12-20 08:06:01Z pedro $
```

---