



Curso: Sistemas Operativos

Profesora: Mireya Paredes

Reporte 3: UNIX Shell and History Feature

Miembros del equipo:

Rafael Agustín López Hernández – 160805

Alexander Díaz Ruiz - 160046

Jose Ashamat Jaimes Saavedra- 158320

Introducción

Shell es un programa que toma comandos del usuario y los ejecuta. Las shell son emuladores de terminales de UNIX llamadas CLIs (Command Line Interfaces), las que se usaban anteriormente de las GUIs (Graphic User Interfaces), que son más versátiles y fáciles de utilizar (*Learning the shell—Lesson 1: What is the shell?*, n.d.).

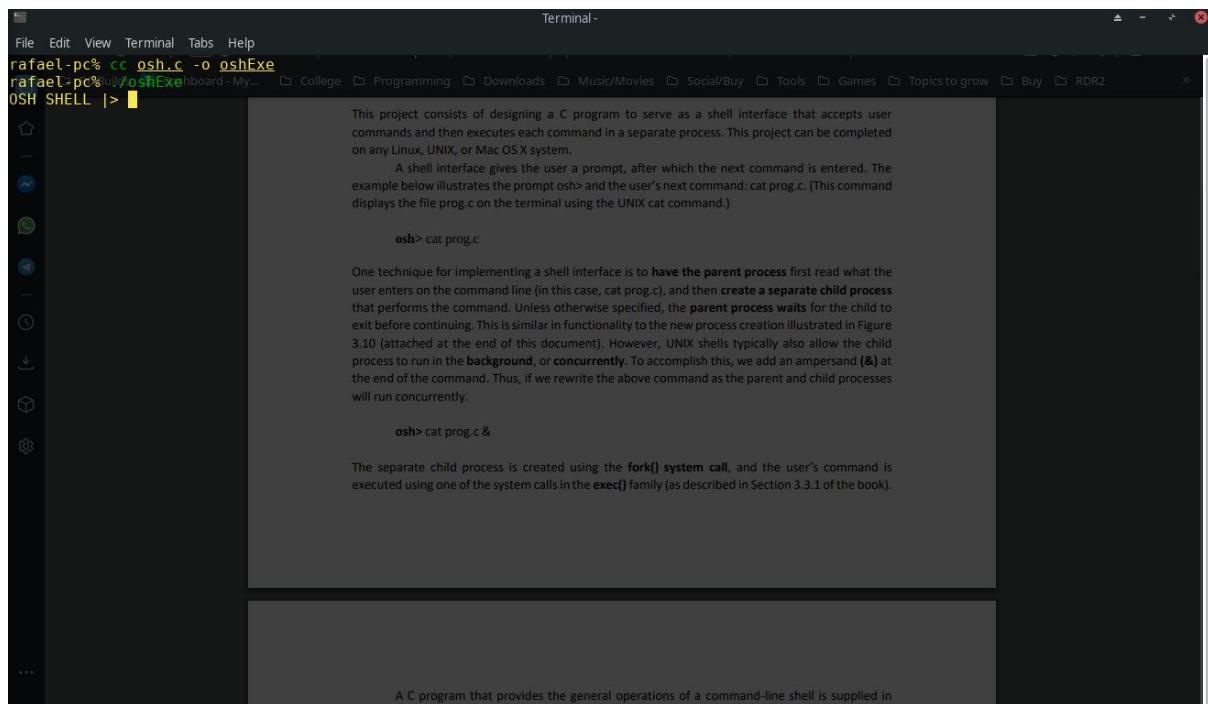
En esta práctica, se hizo un pequeño Shell que se llama “OSH”, que hace lo de una shell normal (comandos de linux), de igual manera tiene un comando para cerrar y un comando para tener el historial de comandos usados.

Parte I

En la primera parte se crea OSH, una shell que acepta comandos que un usuario da. Esta shell podrá usar cualquier comando de otra shell (sin los alias). El uso de comando dentro de OSH es:

- “bye” -> cerrar terminal.
- “history” -> historial de comandos (Se explicará más adelante).

Para iniciar OSH, se corre el programa con el comando “cc osh.c -o oshExe” para compilar y crear el archivo OUT, que será el ejecutable. Para correr el ejecutable, se usa el comando “./oshExe” e inicia OSH.



CÓDIGO DE LA PRÁCTICA

1. `#include <stdio.h>`
2. `#include <unistd.h>`
3. `#define MAX_LINE 80 /* The maximum length command */`
4. `int main(void) {`
 - 4.1. `char *args[MAX_LINE/2 + 1]; /* command line arguments */`
 - 4.2. `int should_run = 1; /* flag to determine when to exit program */`
 - 4.3. `while (should_run) {`
 - 4.3.1. `printf("osh>");`
 - 4.3.2. `fflush(stdout);`
 - 4.3.3. `scanf("%c %c %c", args[0], args[1], args[2]);`
 - 4.3.4. `int pid;`
 - 4.3.5. `pid=fork();`
 - 4.3.6. `if (pid<0){`
 - 4.3.6.1. `printf("Error al crear proceso hijo \n");`

```

        4.3.6.2.    exit();
    4.3.7.    }
    4.3.8.    if(pid==1){
        4.3.8.1.    execvp(args[0],args[1], args[2]);
    4.3.9.    }else if(pid==0 && args[0]=='&'){
        4.3.9.1.    wait();
    4.3.10.    }
    4.3.11.
    4.4.    }
    4.5.    return 0;
5.    }

```

EXPLICACIÓN DEL CÓDIGO

Lineas 1-2; Se incluyen la librería necesaria para el uso de fork(), execvp(), y la librería estándar de entrada y salida.

Linea 3; Se crea una macro para definir el tamaño máximo del comando, en este caso, 80 caracteres.

Linea 4; Inicia el programa principal.

Linea 4.1; Se crea un arreglo de punteros tipo caracteres, para recibir los comandos.

Linea 4.2; Se declara una variable que su valor no cambiará; esto para poder hacer un ciclo infinito y el programa no deje de correr.

Linea 4.3; Se crea un ciclo tipo while, ya que el valor que se encuentra en su parámetro siempre sera "1", entonces el ciclo es infinito.

Linea 4.3.1; Se muestra al usuario el mensaje "osh>".

Linea 4.3.2; Se realiza la limpieza del buffer de salida (stdout).

Linea 4.3.3; Se lee el comando y los parámetros, separados por " ".

Linea 4.3.4; Se crea una variable entera de nombre pid.

Linea 4.3.5; Se crea un proceso hijo con fork(), el valor de fork es almacenado en una variable llamada pid, la función fork() regresa enteros, si el entero es menor a 0 entonces hubo un error al crear el proceso hijo, si es 0 entonces el proceso que está corriendo es el proceso hijo, si el valor es 1 entonces el proceso que corre es el padre.

Lineas 4.3.6-4.3.7; Se crea la validación por si hubo algún problema al crear el proceso hijo.

Linea 4.3.8; Si el valor de pid es 0, entonces:

Linea 4.3.8.1; Ejecuta el comando almacenado en args[0], con los parámetros args[1] y/o args[2], en caso de que no se ingrese un segundo parametro, args[2] será NULL.

Linea 4.3.9; Se crea un else if, para validar si el proceso es el padre y si el comando ingresado es "&".

Linea 4.3.9.1; Si el comando ingresado es "&", entonces el proceso padre ejecutará wait();

Linea 4.3.10; Fin de else if.

Linea 4.3.11; Linea en blanco.

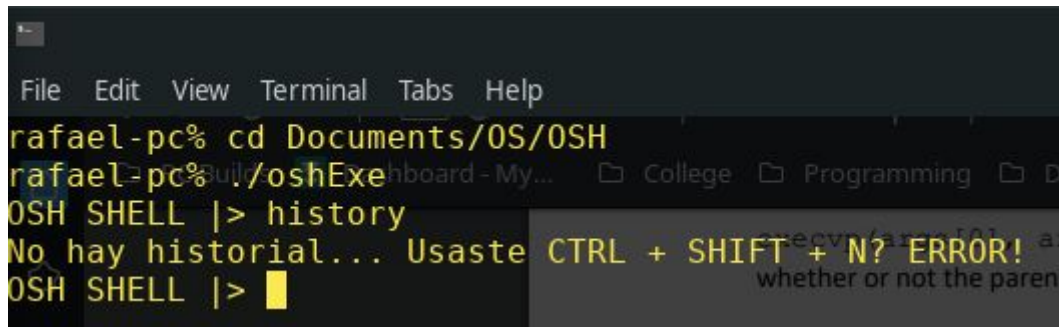
Linea 4.4; Fin del ciclo while.

Linea 4.5; Al final, si el programa corre sin errores debe regresar 0;

Linea 5; Fin del programa principal.

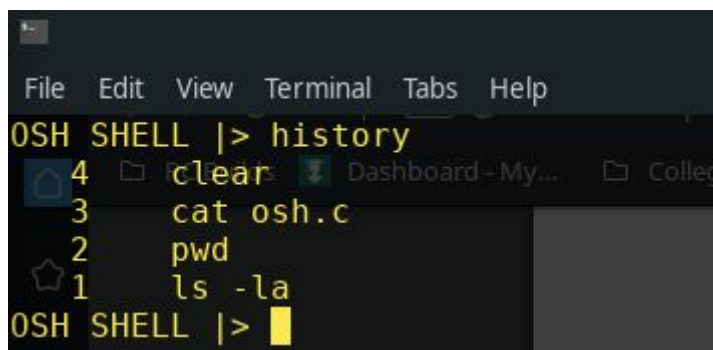
Parte II

La parte dos es tener una manera de tener un historial de comandos que el usuario escribió. Como se dijo en la parte anterior, el comando para el historial es "history". El historial va desde el primero que se puso (toma el lugar 1) hasta el último comando.

A screenshot of a terminal window with a dark background. The menu bar at the top includes 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The prompt is 'rafael-pc%'. The user has navigated to 'Documents/OS/OSH' and entered 'oshExe'. The prompt is now 'OSH SHELL |>'. The user has entered 'history', and the output is 'No hay historial... Usaste CTRL + SHIFT + N? ERROR!'. The prompt is now 'OSH SHELL |>' with a yellow cursor.

```
File Edit View Terminal Tabs Help
rafael-pc% cd Documents/OS/OSH
rafael-pc% ./oshExe
OSH SHELL |> history
No hay historial... Usaste CTRL + SHIFT + N? ERROR!
OSH SHELL |>
```

Si se usa el comando antes de escribir un comando, sale esto:

A screenshot of a terminal window with a dark background. The menu bar at the top includes 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The prompt is 'OSH SHELL |>'. The user has entered 'history', and the output is a list of commands: '4 clear', '3 cat osh.c', '2 pwd', and '1 ls -la'. The prompt is now 'OSH SHELL |>' with a yellow cursor.

```
File Edit View Terminal Tabs Help
OSH SHELL |> history
4 clear
3 cat osh.c
2 pwd
1 ls -la
OSH SHELL |>
```

Cuando hay comandos guardados.

Código:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <string.h>
5.  #include <sys/wait.h>

6.  #define MAX_LINE 80

7.  void invoke_error(char *a)
8.  {
9.      8.1.    printf("%s ERROR! \n", a);
10. }

10. int fetch_input(char *a)
11. {
11.1.    char p;
11.2.    int num = 0;
11.3.    while (((p = getchar()) != '\n') && (num < MAX_LINE))
```

```

11.4.    {
        11.4.1.    a[num++] = p;
11.5.    }
11.6.    if (num == MAX_LINE && p != '\n')
11.7.    {
        11.7.1.    invoke_error("Comando excede largo maximo");
        11.7.2.    num = -1;
11.8.    }
11.9.    else
11.10.   {
        11.10.1.   a[num] = 0;
11.11.   }
11.12.   while (p != '\n')
        11.12.1.   p = getchar();
11.13.   return num;
12.    }

13.    void print_history(char history[][MAX_LINE + 1], int history_count)
14.    {
        14.1.    if (history_count == 0)
        14.2.    {
            14.2.1.    invoke_error("No hay historial... Usaste CTRL + SHIFT + N?");
            14.2.2.    return;
        14.3.    }

        14.4.    int i, j = 10;
        14.5.    for (i = history_count; i > 0 && j > 0; i--, j--)
        14.6.    {
            14.6.1.    printf("%4d\t%s\n", i, history[i % 10]);
        14.7.    }
15.    }

16.    int parse(char *buffer, int length, char* args[])
17.    {
        17.1.    int args_num = 0, last = -1, i;
        17.2.    args[0] = NULL;
        17.3.    for (i = 0; i <= length; ++i)
        17.4.    {
            17.4.1.    if (buffer[i] != ' ' && buffer[i] != '\t' && buffer[i])
            17.4.2.    {
                17.4.2.1.    continue;
            17.4.3.    }
            17.4.4.    else
            17.4.5.    {
                17.4.5.1.    if (last != i-1)
                17.4.5.2.    {

```

```

17.4.5.2.1.    args[args_num] = (char*)malloc(i-last);
17.4.5.2.2.    if (args[args_num] == NULL)
17.4.5.2.3.    {
17.4.5.2.3.1.    invoke_error("No se pudo alocar memoria!");
17.4.5.2.3.2.    return 1;
17.4.5.2.4.    }
17.4.5.2.5.    memcpy(args[args_num], &buffer[last+1], i-last-1);
17.4.5.2.6.    args[args_num][i-last] = 0;
17.4.5.2.7.    args[++args_num] = NULL;
17.4.5.3.    }
17.4.5.4.    last = i;
17.4.6.    }
17.5.    }
17.6.    return args_num;
18.    }

19.    int to_number(char *a)
20.    {
20.1.    int length = strlen(a), i, answer = 0;
20.2.    for (i = 0; i < length; i++)
20.3.    {
20.3.1.    if (a[i] > '9' || a[i] < '0')
20.3.1.1.    return -1;
20.3.2.    answer = answer * 10 + a[i] - '0';
20.4.    }
20.5.    return answer;
21.    }

22.    int main(void)
23.    {
23.1.    char *args[MAX_LINE/2 + 1];
23.2.    int should_run = 1;
23.3.    char history[10][MAX_LINE + 1];
23.4.    int history_count = 0;
23.5.    char buffer[MAX_LINE + 1];
23.6.    memset(buffer, 0, sizeof(buffer));
23.7.    int length, args_num;
23.8.    while (should_run)
23.9.    {
23.9.1.    printf("OSH SHELL |> ");
23.9.2.    fflush(stdout);
23.9.3.    length = fetch_input(buffer);
23.9.4.    if (length == -1)
23.9.4.1.    continue;
23.9.5.    if (strcmp(buffer, "!!!") == 0)
23.9.6.    {

```

```

23.9.6.1.    if (history_count > 0)
23.9.6.2.    {
23.9.6.2.1.    memcpy(buffer, history[history_count % 10],
                MAX_LINE + 1);
23.9.6.2.2.    length = strlen(buffer);
23.9.6.3.    }
23.9.6.4.    else
23.9.6.5.    {
23.9.6.5.1.    invoke_error("No hay comandos en historial...
                PRIVATE BROWSING?");
23.9.6.5.2.    continue;
23.9.6.6.    }
23.9.7.    }
23.9.8.    args_num = parse(buffer, length, args);
23.9.9.    if (args_num == 0)
23.9.9.1.    continue;
23.9.10.   if (strcmp(args[0], "!") == 0)
23.9.11.   {
23.9.11.1.   int temp = to_number(args[1]);
23.9.11.2.   if (temp <= 0 || temp < history_count - 9 || temp >
                history_count)
23.9.11.3.   {
23.9.11.3.1.   invoke_error("No esta ese comando en historial!");
23.9.11.3.2.   continue;
23.9.11.4.   }
23.9.11.5.   else
23.9.11.6.   {
23.9.11.6.1.   memcpy(buffer, history[temp % 10], MAX_LINE + 1);
23.9.11.6.2.   length = strlen(buffer);
23.9.11.6.3.   args_num = parse(buffer, length, args);
23.9.11.7.   }
23.9.12.   }
23.9.13.   if (strcmp(args[0], "bye") == 0)
23.9.14.   {
23.9.14.1.   should_run = 0;
23.9.14.2.   continue;
23.9.15.   }
23.9.16.   if (strcmp(args[0], "history") == 0)
23.9.17.   {
23.9.17.1.   print_history(history, history_count);
23.9.17.2.   continue;
23.9.18.   }
23.9.19.   history_count++;
23.9.20.   memcpy(history[history_count% 10], buffer, MAX_LINE + 1);
23.9.21.   int background = 0;
23.9.22.   if (strcmp(args[args_num-1], "&") == 0)

```



```

23.9.23.  {
           23.9.23.1.  background = 1;
           23.9.23.2.  args[--args_num] = NULL;
23.9.24.  }
23.9.25.  pid_t pid = fork();
23.9.26.  if (pid < 0)
23.9.27.  {
           23.9.27.1.  invoke_error("Fork: Error de proceso!");
           23.9.27.2.  return 1;
23.9.28.  }
23.9.29.  int status;
23.9.30.  if (pid == 0)
23.9.31.  {
           23.9.31.1.  status = execvp(args[0], args);
           23.9.31.2.  if (status == -1)
           23.9.31.3.  {
               23.9.31.3.1.  invoke_error("Error: No se ejecuto el proceso");
           23.9.31.4.  }
           23.9.31.5.  return 0;
23.9.32.  }
23.9.33.  else
23.9.34.  {
           23.9.34.1.  if (background)
           23.9.34.2.  {
               23.9.34.2.1.  printf("PID: #%d corriendo en el background: %s\n",
                               pid, buffer);
           23.9.34.3.  }
           23.9.34.4.  else
           23.9.34.5.  {
               23.9.34.5.1.  wait(&status);
           23.9.34.6.  }
23.9.35.  }
23.9.36.  /**
23.9.37.  * After reading user input, the steps are:
23.9.38.  * (1) fork a child process using fork()
23.9.39.  * (2) the child process will invoke execvp()
23.9.40.  * (3) if command included &, parent will invoke wait()
23.9.41.  */
23.10.  }
23.11.  return 0;
24.  }

```

Explicación del Código:

Líneas 1 - 5: Librerías importadas.

Línea 6: Se define una constante de nombre "MAX_LINE" con valor de 80, el cual hace referencia al mayor número de caracteres permitidos por comando de terminal.

Líneas 7 - 8.1: Se declara la función *invoke_error()*, el cual acepta un puntero de caracteres <a> como parámetro y no regresa una salida.

Al ser llamado, *invoke_error()* imprime el parámetro que le es otorgado (<a>), seguido de "ERROR!" y un salto de línea.

Líneas 10 - 11.2: Se define la función *fetch_input()*, el cual también recibe como parámetro un puntero de caracteres <a>, y regresa el número de caracteres del cual consiste la cadena a la cual apunta.

fetch_input() contiene como atributos un carácter <p> y un entero <num>, inicializado con un valor de 0.

Línea 11.3 - 11.5: Toma lugar un ciclo **while**, el cual ejecuta su contenido siempre y cuando <p> (al cual le es asignado el último carácter ingresado por el usuario) no sea un salto de línea, y <num> sea menor a 80 (MAX_LINE).

Si la condiciones previas son verdaderas, entonces durante cada ciclo, <num> incrementa de valor, y a la cadena <a> le es asignada <p> en la posición <num>.

Es decir, <a> equivaldrá a la entrada que teclee el usuario siempre y cuando no incluya un salto de línea (Enter) y la longitud del comando no exceda los 79 caracteres.

Líneas 11.6 - 11.8: En caso de que el usuario hubiese provisto un comando de 80 caracteres de longitud--de los cuales ninguno fuese un salto de línea--entonces se llamará a la función *invoke_error()* para que despliegue el mensaje "Comando excede largo maximo", seguido por la asignación de (-1) a <num>.

Líneas 11.9 - 11.11: De no ser así--es decir, si el usuario ingresó un Enter como parte de su comando sin haber excedido los 79 caracteres--entonces se concatenará un NULL al final de la entrada del usuario (<a>), para denotar el final de ésta.

Líneas 11.12, 11.12.1: Se ejecuta otro ciclo **while**, en el cual a <p> le será asignado el último carácter ingresado por el usuario siempre y cuando éste no sea un salto de línea.

Línea 11.13: Al final de la función *fetch_input()*, se regresará <num>: la longitud de caracteres en <a>.

En caso de que el comando fuese de longitud inválida (superior a los 79 caracteres), la función regresará (-1).

Línea 13: Se define una función llamada *print_history()*, la cual requiere dos parámetros:

- Un arreglo de cadenas de caracteres llamado <history>, en el cual cada cadena de caracteres dispone de 81 espacios disponibles (MAX_LINE + 1).
- Un entero conocido como <history_count>.

print_history() no regresa una salida.

Líneas 14.1 - 14.3: En caso de que *<history_count>* equivalga a 0, se llamará a la función *invoke_error()* para imprimir el mensaje "No hay historial... Usaste CTRL + SHIFT + N?", y *print_history()* concluirá.

Línea 14.4: Se definen dos enteros: *<i>* y *<j>*; este último inicializado con una valor de 10.

Líneas 14.5 - 15: Se define un ciclo **for**, el cual se ejecutará siempre y cuando *<i>* (ahora equivalente a *<history_count>*) y *<j>* sean mayores a 0; con cada iteración se irán reduciendo los valores de cada entero.

Es decir, el ciclo se repetirá por lo menos *<history_count>* cantidad de veces (el número de comandos previamente escritos), y a lo mucho 10 (*<j>*).

Dentro del ciclo, se imprimirá el valor actual de *<i>* (en caso de que *<i>* conste de menos de 4 dígitos, la diferencia será el número de espacios que le antecedan al ser impreso), seguido de un tabulador, y finalmente el arreglo de caracteres almacenado en la posición [*<i>* MOD 10] de *<history>*.

[*<i>* MOD 10] para cualquier entero positivo *<i>* (dado que *<i>*, inicialmente equivaliendo a *<history_count>*, no puede ser negativo) siempre devolverá un número entre 0 y 10; y ya que *<i>* se reduce con cada ciclo, el valor producido por [*<i>* MOD 10] también se irá reduciendo.

En otras palabras, *<history>* almacenará únicamente los últimos 10 comandos ingresados en el SHELL, los cuales desplegará en orden inverso a través de *print_history()*.

Línea 16: Se declara la función *parse()*, la cual acepta 3 parámetros:

- Un puntero de caracteres llamado *<buffer>*.
- Un entero *<length>*
- Un arreglo de punteros de caracteres *<args>*

, y devuelve un entero como salida.

Líneas 17.1, 17.2: Se definen 3 enteros como atributos de la función:

- *<args_num>*: número de caracteres de tipo espacio, tabulador, o nulo (NULL)--"caracteres inválidos", para propósitos de este análisis--en *<buffer>*. Inicializado con 0.
- *<last>*: índice que señala la posición del carácter inválido en *<buffer>* más reciente. Inicializado con -1, para denotar que el índice está fuera de rango (*<last>* no puede ser inicializado con 0, ya que estaría señalando que el primer carácter de *<buffer>* siempre será inválido).
- *<i>*: índice del carácter actual en *<buffer>*. Inicializado con 0.

Asimismo, el primer elemento del arreglo *<args>* se inicializa con valor nulo.

Línea 17.3: Se declara un ciclo **for**, en el cual el índice `<i>` irá aumentando de valor con cada iteración. El ciclo se seguirá repitiendo hasta que `<i>` sea mayor que `<length>`.

Por ende, el ciclo se ejecutará $(\text{length} + 1)$ veces--es decir, una posición de más de la longitud de la cadena de caracteres a la cual apunta `<buffer>`--debido a que el ciclo tiene el propósito de encontrar por lo menos un carácter inválido en `<buffer>`, aunque tenga que ser el valor nulo encontrado en la posición siguiente del final de cualquier cadena.

Líneas 17.4.1 - 17.4.3: Si el elemento `<i>` del puntero de caracteres `<buffer>` no es un carácter inválido, entonces la ejecución se saltará a la siguiente iteración del ciclo.

Líneas 17.4.4 - 17.4.5.2.4: En caso de que `<i>` esté apuntando a un carácter inválido, y si `<last>` es diferente de (<i> - 1) --en otras palabras, si el más reciente índice de un carácter inválido no fue el índice anterior--entonces se le asignará a `<args>` un puntero de caracteres en la posición `<args_num>`, el cual hace referencia a un bloque de memoria asignada de tamaño (<i> - <last>) : el número de caracteres entre el último carácter inválido y el índice actual.

En caso de que el puntero sea nulo, se invocará a la función `invoke_error()` para imprimir "No se pudo alocar memoria!", y `parse()` regresará el valor de (1).

Dicho de otra manera, con cada iteración del ciclo **for**, se evaluará que la cadena de caracteres a la cual apunta `<buffer>` no contenga caracteres inválidos. De no ser así, entonces en caso de que el carácter inválido no se encuentre al principio de `<buffer>`, o que no haya dos o más caracteres inválidos consecutivos, entonces se almacenará en el arreglo `<args>` un bloque de memoria dinámica, por cada carácter inválido que se encuentre en `<buffer>`. Este bloque tendrá el tamaño necesario como para almacenar cada cadena de caracteres de `<buffer>` en conjunto con su respectivo carácter inválido.

En caso de que el bloque de memoria tenga capacidad nula--lo cual sucede únicamente cuando se intenta asignar memoria de tamaño 0, o excedente a la capacidad disponible--se producirá un mensaje de error, y la función producirá (1) como salida.

Líneas 17.4.5.2.5, 17.4.5.2.6: Acto seguido, se copiarán $(\text{<i> - <last> - 1})$ número de caracteres de la posición $[\text{<last> + 1}]$ de la dirección de `<buffer>`, y se almacenarán en la posición `<args_num>` de `<args>`. De aquellos caracteres copiados, se les concatenará un carácter NULL al final.

Es decir, anteriormente se alojó en la posición `<args_num>` de `<args>` la memoria suficiente como para almacenar una cadena de caracteres (incluyendo su carácter inválido) de `<buffer>`, por cada carácter inválido presente en él. Ahora, aquella memoria se estará empleando para guardar la cadena de caracteres en sí, pero en lugar de conservar el carácter inválido al final, se reemplazará por un NULL para denotar el final de la cadena.

Líneas 17.4.5.2.7 - 18: Una vez que el elemento número `<args_num>` de `<args>` haya sido reasignado--ahora almacenando una cadena de caracteres en lugar de un bloque de

memoria--<args_num> aumenta de valor, y cuyo espacio en <args> almacena un valor NULL.

En otras palabras, <args> termina por almacenar en cada una de sus posiciones los segmentos de <buffer> que no contengan caracteres inválidos--salvo por el último elemento de <args>, el cual contiene NULL.

Y dado que <i> necesitaba estar apuntando a un carácter inválido para ingresar a la condición de la *línea 17.4.5.1*, a <last> se le asigna el mismo índice que <i> por ser el índice del carácter inválido más reciente por el cual ha transcurrido el ciclo.

Finalmente, la función regresa <args_num>: el número de cadenas de caracteres en <buffer> que se encuentran separados por caracteres inválidos.

Líneas 19 - 20.1: Se declara la función *to_number()*, el cual acepta un puntero de caracteres <a> como parámetro, y devuelve un entero--los caracteres de <a> convertidos a un número entero--como salida.

to_number() contiene 3 enteros como atributos:

- <length>: la longitud del arreglo de caracteres al cual apunta <a>.
- <i>: índice del carácter a evaluar en <a>.
- <answer>: número resultante de la conversión de cadena de caracteres <a> a entero. Inicializado con 0.

Líneas 20.2 - 21: Se presenta un ciclo **for**, el cual se ejecutará <length> número de veces. En cada iteración del ciclo, se verificará que el carácter correspondiente de <a> (aquel carácter en la posición <i>) sea estrictamente numérico--es decir, que se encuentre en el rango entre '0' y '9'. Con que un sólo carácter de todo el arreglo de <a> sea alfanumérico--abarcando tanto caracteres alfabéticos como de puntuación--la función regresará un (-1), para señalar que <a> apunta a una cadena inválida para ser convertida a número.

Para hacer la conversión del puntero de caracteres <a> a un número entero, se tendrá que multiplicar el resultado de la conversión hasta el momento (<answer>) por 10 (para denotar una nueva posición decimal), por cada iteración del ciclo; a la cifra resultante se le añadirá el número que resulte de la resta entre el <i>-ésimo carácter de <a> y el carácter '0'.

Cabe mencionar que el valor de un carácter numérico no es el mismo que el dígito que representa, sino el valor decimal de su código ASCII:

- El carácter '0' tiene un valor numérico de 48, en lugar de 0.
- '1' tiene un valor numérico de 49, en lugar de 1.

Y el patrón se repite sucesivamente hasta el '9', con valor de 57.

En ese entonces, si se fuese a restar el carácter '6' menos el carácter '0', por ejemplo, en realidad se estarían restando los valores de sus códigos ASCII: 54 - 48

Lo que resultaría en 6: el valor del carácter al cual se le restó el carácter '0'.

Una vez que se hayan completado todas las iteraciones del ciclo **for**, *to_number()* regresará la cadena de caracteres que le fue entregada como parámetro (<a>), ahora convertida al número entero que representa.

Líneas 22 - 23.7: Se declara la función *main()* del programa, el cual contiene los siguientes atributos:

- Un arreglo de punteros de carácter, <args>, con espacio suficiente para almacenar 41 elementos ($\text{MAX_LINE}/2 + 1$).
- 4 enteros:
 - <should_run>, el cual indica el estado de ejecución del SHELL. Inicializado en 1, para denotar que el SHELL se encuentra activo.
 - <history_count>: el número de comandos previamente ingresados al SHELL, los cuales están guardados en su historial, <history>. Inicializado en 0.
 - <length>: la longitud del comando ingresado en el SHELL.
 - <args_num>: el número de cadenas de caracteres en un comando, separados por caracteres inválidos (espacios, tabuladores, valores de tipo NULL).
- Un arreglo de cadenas de caracteres bajo el nombre de <history>. Es aquí en donde se guardan los comandos previamente ingresados al SHELL. <history> permite almacenar hasta 10 comandos, cada uno de 81 caracteres de largo ($\text{MAX_LINE} + 1$).
- <buffer>: un arreglo de 81 caracteres ($\text{MAX_LINE} + 1$). Representa el comando que se ha ingresado a la terminal. Sus 81 elementos están inicializados con ceros.

Líneas 23.8 - 23.9.2: *main()* emplea un ciclo **while** para su ejecución: mientras <should_run> se mantenga en 1, el SHELL se mantendrá corriendo.

Lo primero que hace el programa es imprimir "OSH SHELL |> ", para indicar un nuevo prompt o línea de comando. Se emplea la función *fflush()* para garantizar que el prompt se mantenga visible, antes de bloquear para recibir entrada del usuario--sin ello, es posible que la pantalla se quede en blanco.

Líneas 23.9.3 - 23.9.7: Se llama a la función *fetch_input()*, otorgándole a <buffer> como argumento. La salida de la función se almacena en la variable <length>.

fetch_input() se encarga de guardar la entrada tecleada a <buffer>, y a producir como salida la longitud de ésta.

Si la cadena que almacena <buffer> supera los 79 caracteres, entonces se considerará como una entrada inválida, y el ciclo **while** saltará a su siguiente iteración.

En caso de que el usuario haya ingresado "!!", se buscará volver a ejecutar el comando ingresado más recientemente a la terminal.

Para ello, en primer lugar, se debe de cerciorar que el usuario haya ingresado comandos con anterioridad. De no ser así, se llamará al método *invoke_error()* para imprimir "No hay comandos en historial... PRIVATE BROWSING?", y el ciclo empezará una nueva iteración.

”

Sólo en caso de que <history_count>--es decir, el número de comandos escritos anteriormente--sea positivo es que <buffer> alojará 81 caracteres ($\text{MAX_LINE} + 1$) del

arreglo <history>. El elemento del cual copiar la memoria dependerá de <history_count>, de tal forma que su MOD 10 determinará el índice del arreglo.

Esto se debe a que <history> almacena únicamente los 10 últimos comandos ingresados al SHELL, por lo que cualquiera que sea el número al que equivalga <history_count>, su módulo se mantendrá en el rango [0 - 9]: los índices de cada elemento de <history>.

Una vez que <buffer> contenga la instrucción recuperada de <history>, se calculará la longitud de su cadena y se almacenará en <length>.

Líneas 23.9.8 - 23.9.9.1: Ahora que <length> y <buffer> se encuentran con valores asignados, es posible invocar a la función *parse()* y registrar su salida en la variable <args_num>.

Cabe recordar que *parse()* se encarga de dividir el comando presente en <buffer> por cada carácter de espacio, tabulador, y carácter nulo presente en la cadena a la cual apunta. Cada fragmento de <buffer> se alojará en un elemento de <args> conforme su orden de aparición, y finalmente *parse()* producirá como salida el número de fragmentos realizados.

En caso de que *parse()* haya devuelto un 0 (lo cual no debería ser posible, porque cada cadena de caracteres en C contiene un carácter nulo al final para denotar dónde termina, pero la condición se añadió de todas maneras como manejo de errores y excepciones), el ciclo ejecutará su siguiente iteración.

Líneas 23.9.10 - 23.9.12: Si el primer elemento de <args>--esto es, el primer fragmento de <buffer>--es un signo de exclamación, entonces se declarará un entero local llamado <temp>, al cual le será asignado la salida de la función *to_number()* con el segundo elemento de <args> como argumento.

<temp> es referente al índice del comando a volver a ejecutar. Tomará el valor representado por el carácter alojado en la segunda posición del arreglo <args>, en caso de que tal carácter sea numérico. De no ser así, entonces <temp> equivaldrá a (-1).

En caso de que el usuario esté intentando acceder a un comando de índice nulo o negativo, o uno anterior a los últimos 10 que han sido ejecutados, u otro con índice aún no ingresado, entonces el SHELL llamará a la función *invoke_error()* para imprimir el mensaje de error "No esta ese comando en historial!", y continuar con la siguiente iteración del ciclo.

De no ser así, y el comando propuesto por el usuario es válido, entonces <buffer> almacenará el comando a ejecutar del elemento número [<temp> MOD 10] de <history>.

A diferencia de la instrucción de la *línea 23.9.6.2.1*, aquí se aplica el módulo 10 sobre <temp> en lugar de <history_count>, ya que se está buscando guardar en <buffer> el comando de índice especificado después del signo de exclamación: <temp>, y no el del comando más reciente, el cual vendría siendo <history_count>.

Acto seguido, <length> alojará la longitud de caracteres presente en el comando de <buffer>, y <args_num> su número de segmentos.

Líneas 23.9.13 - 23.9.18: Existen alternativas para la condición presente en la línea 23.9.10, tal que no necesariamente necesita ser un signo de exclamación el primer elemento de <args>.

En caso de registrar la cadena "bye", a <should_run>--la variable de la cual depende el ciclo **while** que controla la ejecución del programa--le será asignado el número 0, rompiendo el ciclo, y consecuentemente, finalizando el SHELL.

En caso de ser "history", se invocará a la función *print_history()* para imprimir los 10 últimos comandos ingresados a la terminal, junto con sus índices. Inmediatamente después, el programa se saltará a la ejecución del siguiente ciclo.

Líneas 23.9.19 - 23.9.24: Independientemente del comando que haya ingresado el usuario--por excepción de "bye" y "history, ya que éstos saltan a la siguiente iteración del ciclo--el número de comandos ingresados, <history_count>, aumentará.

Acto seguido, se almacenará la instrucción a ejecutar (<buffer>) en el elemento [<history_count> MOD 10] de <history>.

Por último, se declarará un entero local llamado <background>, inicializado con 0, el cual determina si el estado de ejecución de un proceso hijo será concurrente al de su padre o no.

Si la última cadena de caracteres de <buffer>--es decir, el penúltimo elemento de <args> (no el último, debido a que la función *parse()* siempre aloja un carácter nulo en el último elemento de <args>)--hubiese sido un "&", entonces el usuario estará indicando que el proceso hijo a crear, mismo que se encargará de llevar a cabo el comando escrito, correrá concurrentemente con el proceso padre, el cual recupera el comando a ejecutar.

En ese caso, se le asignará un (1) a <background> para señalar ello mismo.

Asimismo, <args_num> decrementará de valor, y a su espacio correspondiente en <args> le será asignado un carácter nulo--en otras palabras, se removerá el "&" del comando a ejecutar.

Líneas 23.9.25 - 23.9.32: Se declara un entero local llamado <status>, y se crea el proceso hijo que ejecutará el comando, el cual es identificado con el ID <pid>.

Si <pid> resulta ser menor a 0, significa que se produjo un error al momento de bifurcar el proceso principal. Se llamará a la función *inovke_error()* para imprimir el mensaje "Fork: Error de proceso!", y concluirá la ejecución del SHELL.

En caso de que <pid> sea igual a 0, entonces estará referenciando al proceso hijo.

Se invocará la función *execvp()*, a la cual se le otorgarán el primer elemento de <args> y <args>, respectivamente, como sus argumentos. La salida de la función se guardará en <status>.

execvp() es una función predefinida en C que ejecuta el comando presente en su primer argumento, y emplea el arreglo que le es otorgado como segundo argumento como el listado de argumentos del comando a realizar.

Si la función falla, producirá (-1) como su salida, por lo que el SHELL llamará a la función *invoke_error()* para imprimir "Error: No se ejecuto el proceso", y el programa del proceso hijo concluirá.

Líneas 23.9.33 - 24: Si <pid> hace referencia al proceso principal, entonces se verificará el estado de <background>.

En caso de que el usuario haya permitido la ejecución concurrente de proceso principal y proceso hijo, entonces se imprimirá en la terminal: "PID: #<pid> corriendo en el background: <buffer>".

De no ser así, entonces el proceso principal esperará a que el proceso hijo concluya su operación antes de resumir ejecución, y a <status> le será asignado el valor que regrese el subprograma.

Referencias

C library function - memcpy(). (2020). tutorialspoint. Recuperado el 15 de marzo de 2020, de

https://www.tutorialspoint.com/c_standard_library/c_function_memcpy.htm

continue statement in C. (2020). tutorialspoint. Recuperado el 15 de marzo de 2020, de

https://www.tutorialspoint.com/cprogramming/c_continue_statement.htm

C strlen(). (2020). Programiz. Recuperado el 15 de marzo de 2020, de

<https://www.programiz.com/c-programming/library-function/string.h/strlen>

Learning the shell—Lesson 1: What is the shell? (s.f.). Recuperado el 6 de febrero de 2020,

de http://linuxcommand.org/lc3_lts0010.php

memset() in C with examples. (s.f.). GeeksforGeeks. Recuperado el 15 de marzo de 2020,

de <https://www.geeksforgeeks.org/memset-c-example/>

Microsoft. (4 de noviembre del 2016). *return Statement (C)*. Recuperado el 15 de marzo de

2020, de

<https://docs.microsoft.com/en-us/cpp/c-language/return-statement-c?view=vs-2019>

Prabhu, R. (s.f.). *Dynamic Memory Allocation in C using malloc(), calloc(), free() and*

realloc(). Recuperado el 15 de marzo de 2020, de

<https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>