

## FICHA RESUMEN EJERCICIO ENTREGABLE: IMPLEMENTACIÓN DE UN SHELL CORREGIDO

- **Esquema inicial:**

En la primera clase nos dedicamos a realizar un esquema en papel del que partir para este ejercicio. El mío es el siguiente:

```

main
{
    i=1; while(i==1)
    {
        Declaración
        instuccion = instuccion
        ejecutable = instuccion [0]
        if ("exit" || "quit")
        {
            exit(0);
            i=0;
        }
        else
        {
            ejecucion = fork();
            // crea proceso waitpid (ejecucion, status)
            // ejecucion es igual a 0 entonces;
            execvp ( ejecutable , (instuccion+i) );
            exit(0);
        }
        // Toda esto vuelve al padre:
        Padre {
            waitpid ( ejecucion , (WIFEXITED) , 0 );
        }
    }
}

```

Mi idea inicial era que primero debía leer una línea del usuario, que debía separar según los espacios que tenga. También debía de comprobar si el primer elemento de la línea es exit para finalizar el Shell. Debemos tener una condición para comprobar si estamos en el proceso hijo o padre. El hijo tiene el pid declarado igual a 0 y debe ejecutar con execvp que recibe como parámetros el primer elemento y el vector que contiene todos

los elementos de la instrucción. Si estamos en el padre debemos usar `waitpid` para esperar por la finalización del proceso.

Esta era la idea inicial que se fue transformando a medida que avanzaba en el programa.

- **Desarrollo del programa:**

Para comenzar el desarrollo de esta actividad me centre en los ejemplos que se encontraban en el campus virtual. En primer lugar, me fije en el de `strtok` (usando Python tutor también), que sería la función que debíamos utilizar para separar la string que iba introducir el usuario en distintos bloques. También observe como se comportaba la propia terminal y que estructura seguía, siendo la siguiente: "usuario@administrador:~ruta\$ ". Por lo que también debía buscar información en internet sobre cómo obtener estos nombres.

Durante buena parte del trabajo he intentado crear una función paralela `parse()` para separar la línea según los espacios, pero como no lo he conseguido lo he incluido en la implementación del propio Shell, sin embargo antes voy a presentar un pequeño archivo que contiene funciones útiles para el funcionamiento de nuestro Shell.

- **Archivos:**

- **Funciones.c:** En este archivo se incluyen dos funciones que permiten un mejor funcionamiento del Shell. La primera de ellas es `char *minuscula(char *S1)`, que nos permite introducir nuestros comandos sin distinción de mayúsculas o minúsculas. Su funcionamiento es simple, debemos pasar como parámetro un puntero `s1`, para recorrerlo usaremos un bucle que comienza con `j = 0`, y no se detiene hasta que `j` llegue al valor de `len` siendo este la longitud de la string a la que apunta el puntero `s1`. A este valor accedemos con la función `strlen()` de `string.h`. Durante el bucle `j` se incrementará en uno con cada iteración y sobre el carácter al que apunta la string aplicaremos la función `tolower`, de `ctype.h` que convierte los caracteres en minúsculas. La otra función es `salida()` que hace de interfaz gráfica de nuestro terminal. Es una función de tipo `void` que únicamente imprime la cabecera de la línea. Recordemos que para emularla necesitamos el nombre de usuario, el del administrador del equipo y la ruta de directorios en la que nos encontramos. Para obtener el usuario usamos un puntero llamado `user` al que se le asigna el valor de la función `getlogin()`, del paquete `unistd.h` que devuelve el usuario actual. La ruta de directorios se guarda en el vector de caracteres `cwd` de con un tamaño máximo de 1024, la función `getcwd` de `unistd.h` permite obtener el valor de la ruta de datos actual. Para ello, debemos pasar como primer parámetro el vector y como segundo parámetro su tamaño. Por último, necesitamos el administrador que se guardará el vector de caracteres `hostname` de tamaño 128. Seguimos un procedimiento similar al de la ruta de directorios, pero con la función `gethostname()` de `unistd.h`. Para mostrar la interfaz usamos la instrucción `printf("%s@%s:~%s$ ", user, hostname, cwd)`. Su código es el siguiente:

```
#include <unistd.h> // Incluye la librería unistd.h de donde usaremos la función
chdir() para cambiar de directorio, así como la operación getpid que nos
devuelve la etiqueta del proceso hijo, además nos da los
```

nombres de usuario, administrador y la ruta de directorios que usaremos en este archivo.

```
#include <string.h> //Incluye la librería string.h que nos proporciona las funciones strtok para separar por espacios una string, fgets() para poder leer una string por teclado y strcmp() para comparar una string con otra.
```

```
#include <ctype.h> //Incluye la librería ctype.h que contiene la función tolower() que será usada en funciones.c, archivo que a su vez será importado con la cabecera funciones.h.
```

```
#include <stdio.h> //Incluye la librería stdio.h que proporciona algunas funciones básicas.
```

```
#include "funciones.h" //Incluye la cabecera funciones.h que proporciona dos funciones simples, char* minuscula(char *s1) y void salida(), que se han implementado en el archivo fuente funciones.c.
```

```
char* minuscula(char *s1){ //Defino la función minuscula que devuelve un puntero a caracteres y recibe como parámetro un puntero de caracteres(*s1). Va a devolver la string apuntada por el puntero pero convertida a letras minúsculas.
```

```
    len = strlen(s1); //Declara entero llamado len que almacenará el valor devuelto por la función strlen() sobre el puntero pasado como parámetro, siendo este valor la longitud de la string en s1.
```

```
    for(j = 0; j < len; j++){ //Bucle que comienza en j = 0, e y sigue iterándose mientras sea menor que len, con cada iteración j se incrementa en 1.
```

```
        *s1 = tolower(*s1); //Cada elemento apuntado por j va a convertirse en minúscula haciendo uso de la función tolower() de ctype.h.
```

```
        s1++; //Apuntamos al siguiente elemento de la string.
```

```
    }
```

```
    s1 -= len; //Al finalizar, debemos retroceder al primer elemento al que apuntaba el puntero.
```

```
    return s1; //Devolvemos el puntero una vez convertido todos sus elementos a minúscula.
```

```
}
```

```
void salida(){ //Define la función salida de tipo void. No recibe ningún argumento y debe mostrar la línea de comandos.
```

```
user = getlogin(); //Declaramos un puntero a caracteres llamado user, donde se guardará el valor de getlogin() que devuelve el usuario actual y se encuentra en el paquete unistd.h.
```

```
getcwd(cwd, sizeof(cwd)); //Usamos la función getcwd(unistd.h), que recibe como parámetros un puntero a caracteres y el tamaño de ese puntero. En este caso, guardaremos en cwd la ruta de directorio actual.
```

```
gethostname(hostname, sizeof(hostname)); //Usamos la función gethostname(unistd.h), que recibe como parámetros un puntero a caracteres y el tamaño de ese puntero. En este caso guardaremos en ese puntero el valor del usuario administrador.
```

```
printf("%s@%s:~%s$ ", user, hostname, cwd); //Mostramos en pantalla la estructura de entrada del shell, esto es "usuario@administrdor:~ruta$ ", tras este mensaje el usuario debe introducir el comando a ejecutar.
```

```
}
```

- **Funciones.h:** esta es la cabecera que nos va a permitir usar las funciones de funciones.c en nuestro programa principal, además en ella referenciamos las variables usadas en este archivo. Primero tenemos que definir el nombre de la cabecera en caso de que no lo este con `#ifndef` y `#define` `FUNCIONES_H`. Después, declaramos los vectores de caracteres `cwd` y `hostname`, así como el puntero `user`, y la propia función de salida (`printf`). Esta declaración se realiza de manera externa para que cualquier archivo que incluya la cabecera con `#include "funciones.h"` pueda usarla. De la misma manera, declaramos las variables que usa la función `printf`, los enteros `j` y `len`, y la propia función `printf` también de manera externa. Su código es el siguiente:

```
#ifndef FUNCIONES_H //Ejecuta este archivo antes de comprimir.
```

```
#define FUNCIONES_H //Define el la cabecera que será importada por otros archivos.
```

```
//Las variables declaradas sin extern solo se usan en funciones.c.
```

```
char *user; //Declaramos un puntero a caracteres llamado user, donde se guardará el valor de getlogin() que devuelve el usuario actual y se encuentra en el paquete unistd.h.
```

```
char cwd[1024], hostname[128]; //Declaro en primer lugar msg que es un vector de caracteres donde se va a guardar la entrada del ususario en nuestro terminal, el segundo cwd guardará la ruta de datos actual y hostname va a guardar el nombre del administrdor del equipo.
```

`extern void salida(); //Declaramos la función salida que no va a devolver ningún valor, sino que va a imprimir la estructura de nuestra línea en comandos. La declaramos con extern para que cualquier archivo que incluya la cabecera pueda usarla.`

`extern char* minuscula(char *s1); //Declaramos la función minuscula en la cabecera, para que cualquier archivo que la incluya pueda usarla. Esta recibe como parámetro un puntero a caracteres y devuelve un resultado del mismo tipo. La declaramos con extern para que cualquier archivo que incluya la cabecera pueda usarla.`

`int j, len; //Declaramos dos enteros, j para las iteraciones del bucle, y len para guardar la longitud de una string haciendo uso de la función strlen.`

`#endif //Finalizar si la ejecución se ha realizado correctamente.`

- **Programa fuente:**

El archivo shell.c contiene nuestro programa principal. Para facilitar la explicación vamos a dividir en partes el programa:

- **Incluyendo cabeceras:**

Este segmento del código es muy sencillo ya que es donde incluimos las librerías que vamos a usar, en este caso son:

- `#include <sys/types.h>`
- `#include <unistd.h>`
- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <sys/wait.h>`
- `#include <ctype.h>`
- `#include <string.h>`
- `#include "funciones.h"`, esta es la cabecera de las funciones que vamos a importar desde funciones.c.

Además declaramos unas variables globales que nos serán de ayuda para comprobar si el usuario quiere ejecutar algún caso especial(exit/quit o cd), en quit y exit(vectores de caracteres) almacenamos los valores "quit\0" y "exit\0", respectivamente

El Código es el siguiente:

`#include <unistd.h> //Incluye la librería unistd.h de donde usaremos la función chdir() para cambiar de directorio, así como la operación getpid que nos devuelve la etiqueta del proceso hijo.`

`#include <string.h> //Incluye la librería string.h que nos proporciona las funciones strtok para separar por espacios una string, fgets() para poder leer una string por teclado y strcmp() para comparar una string con otra.`

```
#include <ctype.h> //Incluye la librería ctype.h que contiene la función
tolower() que será usada en funciones.c, archivo que a su vez será importado
con la cabecera funciones.h.
```

```
#include <stdio.h> //Incluye la librería stdio.h que proporciona algunas
funciones básicas.
```

```
#include <sys/wait.h> //Incluye la librería sys/wait.h que proporciona las
funciones de la familia wait que nos harán falta.
```

```
#include <sys/types.h> //Incluye la librería sys/types.h que proporciona la
declaración de procesos y permite trabajar con ellos con funciones como fork
y exec.
```

```
#include "funciones.h" //Incluye la cabecera funciones.h que proporciona
dos funciones simples, char* minuscula(char *s1) y void salida(), que se han
implementado en el archivo fuente funciones.c.
```

```
#include <stdlib.h> //Incluye la librería stdlib.h de la que usaremos la función
exit().
```

```
//Declaración variables globales.
```

```
char exitw[5] = "exit\0"; //Declaramos en exitw la cadena de caracteres
"exit\0" de tamaño 5, que servirá para conocer si el usuario quiere detener la
ejecución del shell.
```

```
char quitw[5] = "quit\0"; //Declaramos en quitw la cadena de caracteres
"quit\0" de tamaño 5, que también permitirá conocer si el usuario quiere
detener la ejecución del shell.
```

```
char cdw[3] = "cd\0"; //Declaramos en cdw la cadena de caracteres "cd\0"
de tamaño 3, que será útil para saber si el usuario que realizar la operación
cd(cambiar de directorio).
```

- **Declaración de variables:**

En este apartado declaramos las variables que vamos a usar a lo largo del desarrollo del programa. Necesitamos la etiqueta para un proceso hijo(child\_pid) que se declara con pid\_t, y para controlar su estado necesitamos el entero status que nos proporciona la librería sys/wait.h. Declaramos el entero i con valor 1 como apoyo para crear un bucle while indefinido, que nos permitirá ejecutar tantas veces como deseamos nuestro Shell. En msg declaramos un vector de caracteres de tamaño 1024 que almacenará la entrada del usuario, el entero k nos será útil para comprobar si el comando del usuario es asíncrono (más adelante se explicará), esto será cuando tome el valor 1(inicialmente es 0). Continuando en el apartado de enteros tenemos cont que usaremos como apoyo para referencia a un vector de strings(tira) , y j para saber si es la primera vez que usamos el strtok , teniendo que hacer uso de la función minuscula(ya que es el primer comando del usuario y debe ejecutarse en cualquier forma de

escritura). A su vez, usaremos dos punteros ptr y ejecutable, ptr apuntará a la entrada del usuario y será un parámetro de strtok, ejecutable será donde alojemos el resultado de esta función y sobre el que iteraremos. Por último tenemos tira, que es un vector de strings donde se almacenará cada elemento de la línea de usuario, una vez haya sido separado con strtok(por espacios), y servirá como argumento del execvp.

El código es el siguiente:

```
void main(){  
    pid_t child_pid; //Declaramos una etiqueta de proceso que se llame  
    child_pid que representará al proceso hijo que ejecutará las instrucciones.  
  
    int status; //Declaramos el entero status que será un argumento para  
    esperar por la ejecución del proceso hijo hasta un determinado estado.  
  
    int i = 1; //Declaramos un entero con i con el valor de 1, nos permitirá  
    crear un bucle while que ejecute nuestro shell hasta que se escriba exit o  
    quit.  
  
    char msg[1024]; //Declaramos un vector de caracteres llamado msg  
    que va a ser asignado a la entrada del usuario, con el tamaño máximo de una  
    línea(1024).  
  
    int k = 0; //Declaramos un entero k que nos ayudará a conocer si el  
    usuario ha introducido una instrucción asíncrona, ya que se activará a uno en  
    este caso.  
  
    int cont, j; //Declaro un contador(cont) que servirá como índice en el  
    vector de strings tira, j servirá como contador para informarnos en caso de  
    que sea la primera vez que usamos strtok, teniendo que usar  
    minúscula de funciones.h.  
  
    char *ptr, *ejecutable; //Declaramos dos punteros a caracteres, ptr  
    apuntará a la entrada del usuario, y ejecutable será usado para alojar el  
    resultado de la función strtok, que separará la string  
    apuntada por ptr en espacios.  
  
    char *tira[128]; //Declaramos en tira un vector de strings(128 de  
    tamaño) que servirá para almacenar los elementos separados por espacios  
    de la entrada del usuario. Se pasará como parámetro al exec que ejecutará el  
    proceso hijo.
```

- **Lectura, separar línea de usuario, almacenar y casos especiales:**  
Para comenzar el programa recordemos que la función main es de tipo void porque no necesitamos devolver un valor que finalice la ejecución. Tenemos un bucle “infinito” ya que dura mientras i sea igual a 1, condición que siempre se cumple. Desde aquí saltamos estamos en el “proceso padre”, le damos a el valor 0 para reinicializarlo y tras ello llamamos a la función salida() de

funciones.c que imprime la estructura de una línea de comandos (usuario@administrdor:~ruta\$).

Nuestro shell pedirá la entrada del usuario con la función fgets(msg, 1024, stdin) que guarda en msg una string de máximo tamaño 1024 siguiendo el la entrada estándar(stdin). También reiniciamos los valores de j y cont, y hacemos que ptr apunte a msg.

Es aquí cuando comenzamos a “trocear la línea del usuario”, le asignamos a ejecutable el valor de strtok(ptr, “\t\n\r”). Esta función convierte el primer espacio (“\t\n\r”) al que apunte ptr en ‘\0’ separando una primera palabra a la que apunta ejecutable. Como j es igual a 0 tenemos que convertir la string de ejecutable en minúscula con la función de funciones.c. Tras ello realizamos otro strtok(null, “\t\n\r”), pero con null como argumento para continuar del el elemento al que ejecutable convirtió en ‘\0’, e incrementamos j para salir del primer caso. Seguimos usando strtok mientras ejecutable no sea null, es decir, no haya llegado al fin de la string.

Durante estos dos bloques los valores que toma ejecutable se almacenan en el vector de string tira[cont++] en el que el contador va iterando para ir avanzando en los índices. El último elemento al que referencia tira va a ser “0”, para poder ejecutar el comando en caso de que sea simple (como ls sin argumentos).

Antes de iniciar el proceso hijo, tenemos que comprobar que el usuario no ha escrito ninguna orden que cancele la ejecución del Shell esto se haría con exit o quit. Por ello usamos un if en if (strcmp(tira[0], exitw) == 0 || strcmp(tira[0], quitw) == 0), ya que strcmp(string.h) devuelve 0 si las dos strings son iguales, y recordemos que en tira[0] se almacena el primer elemento de la línea de comandos y, quitw y exitw, hacen referencia a estos comandos especiales. Si se cumple rompemos el bucle “infinito” con break, deteniendo el Shell.

El código es el siguiente:

```
while(i == 1){ //Mientras i sea 1 este bucle va a seguir ejecutándose.

    //Si no estamos en el proceso hijo, debemos ejecutar el
    proceso padre que contiene el cuerpo de nuestro shell.

    k = 0; //Le damos a k el valor a 0, ya que es una nueva
    iteración que debe reiniciar esta variable.

    salida(); //Llamamos a la función salida de la cabecera
    "salida.h", para imprimir la línea de comando de nuestro shell.

    fgets(msg, 1024, stdin); //Leemos el mensaje de que
    introduce el usuario con fgets de string.h, recibe como parámetros el vector
    msg, su tamaño y stdin(standard input).
```



```

        j = 0; //Ponemos el valor de j a 0, al igual que k
porque reiniciamos el valor.

        cont = 0; //Declaro un contador a 0 que servirá como
índice en el vector de strings tira.

        ptr = msg; //El puntero ptr apunta al vector pasado
como parámetro.

        ejecutable = strtok(ptr, "\t\n\r"); //El puntero
ejecutable es igual al valor devuelto por la función strtok(de string.h). Esta
recibe como parámetros ptr, que era la entrada
del usuario y lo que queremos usar
como separador, en este caso "\t\n\r" que representa el espacio. En
ejecutable se guarda el elemento
antes del primer espacio poniendo este espacio a '\0'.

        while( ejecutable != NULL) { //Mientras ejecutable sea
distinto de NULL, se ejecutará este bucle para separar por espacios la entrada
del usuario.

                if(j == 0){ //Si j es igual a 0 significa que es la
primera iteración y por lo tanto debemos darle un trato especial, ya que
antes del primer espacio se escribe el
comando como cd o ls.

                        ejecutable =
minuscula(ejecutable); //Lo primero que hacemos es aplicar sobre el puntero
ejecutable la función minuscula que definimos en funciones.c. Así
podemos escribir
nuestro comando con o sin minúsculas que se ejecutará sin problemas.

                                tira[cont++] = ejecutable;
//Almacenamos el valor del puntero ejecutable en el puntero tira, que
avanza de posición tras la iteración. Recordemos que tira es el
vector que se pasa como parámetro en el
execvp y contiene toda la línea que introduce el usuario, con su elemento
separados.

                                        ejecutable = strtok(NULL, "\t\n\r");
//Realizamos otro strtok para separar con espacios pero en este caso no
usaremos ptr, si no NULL que es el elemento en el
que nos hemos quedado.

                                                j++; //Incrementamos j para saber que
no estamos en el comando principal que introdujo el usuario.

}

```

```

else{ //Si j es distinto de 0 nos encontramos
en el segundo elemento antes de un espacio al que apunte ejecutable.

tira[cont++] = ejecutable; //Volvemos
a almacenar en el puntero tira el elemento separado con espacio, tras eso,
avanzará un elemento.

ejecutable = strtok(NULL, " \t\n\r");
//Realizamos otro strtok para separar con espacios pero en este caso no
usaremos ptr, si no NULL que es el elemento en el
que nos hemos quedado.

}

}

tira[cont++] = 0; //El último elemento del
puntero tira es igual a 0, para permitir la ejecución de instrucciones de un
único elemento(ej. ls sin argumentos).

if (strcmp(tira[0], exitw) == 0 ||
strcmp(tira[0], quitw) == 0){ //En este if usamos la función strcmp del
módulo string.h que recibe como parámetros dos
strings, siendo
tira[0] el primer elemento de la entrada del usuario. Si este es igual a quit o
a
exit, el valor devuelto por la función es 0, en caso
contrario es 1. Si es 0, debemos hacer romper el
bucle con
break.

return -1; //Devolvemos -1 para
finalizar el programa.

}

```

- **Lanzamiento del proceso hijo y cd:**

Si el usuario no ha escrito exit o quit, debemos pasar a ejecutar el comando del usuario. Lo primero es conocer si la operación a realizar es un cambio de directorio(cd) que se debe hacer desde el proceso padre para poder cambiar la ruta del nuestro entorno, si lo hiciéramos en el hijo, solo se guardaría en este siendo inefectivo. Usamos strcmp para conocer si el primer elemento de tira es cd (guardado en cwd), si devuelve 0 es cierto y debemos cambiar de directorio usando chdir(tira [1]), siendo tira[1] el segundo elemento de tira(un comando cd solo tiene dos partes cd y el directorio).

Si no es cd, tenemos dos opciones. Empezaremos por el caso asíncrono, el comando del usuario finaliza en &. Para comprobar si esta es nuestra situación tenemos que hacer unos ajustes. En primer lugar, reducimos cont en 2 para

que tira referencia al último elemento que escribió el usuario ({usuario, "0", '\0'}). Declaramos un entero llamado longitud donde almacenamos la longitud de la string tira[cont]. Esta longitud la usaremos para como índice dentro de la string, ya que tira[cont][longitud-1] es igual al último carácter de la última string del usuario. Comprobamos si este carácter es igual a "&" con un if.

En caso de que lo sea, hacemos que tira[cont][longitud-1] igual a '\0' porque si dejáramos el &, la instrucción no podría ejecutarse, k pasa a ser 1(nos servirá en el proceso hijo). Iniciamos el hijo con fork() y hacemos sleep(0) para hacer que el terminal no espere.

En caso de que la ejecución no sea asíncrona tenemos que llamar a fork() con child\_pid y usar waitpid(child\_pid, &status, 0) para esperar a que el proceso hijo finalicé.

```
else{ //Si no se ha introducido exit o quit.
    if(strcmp(tira[0], cdw) == 0){ //Si el
valor de tira[0] es cd no se debe lanzar el proceso hijo sino que debemos
cambiar el directorio con chdir en el padre.
        chdir(tira[1]); //El elemento
que contiene el directorio es el segundo en el vector tira(tira[1]) que
contiene la instrucción. Por lo que usamos la
        función chdir que permite el cambio de
directorio con este elemento como parámetro.
    }
    else{ //Si no es cd, debemos ejecutar
el proceso hijo.
        cont -= 2; //Antes de ejecutar
el proceso hijo, debemos comprobar que la instrucción introducida no es
asíncrona. Para comenzar reducimos la variable
        cont en 2, para que tira referencia al último
elemento escrito por el usuario({"entrada usuario", "0", '\0'}).
        int longitud =
strlen(tira[cont]); //Creamos un entero llamado longitud que nos da la
longitud de la string a la que apunta tira.
        if(tira[cont][longitud-1] ==
'&'){ //Si este último carácter es igual a '&', el usuario introdujo un comando
asíncrono, y debemos darle un trato
        especial. Tenemos que
imprimir el pid del hijo por pantalla y seguir ejecutando la terminal sin
espera.
        tira[cont][longitud-1] =
'\0'; //Para que el hijo pueda ejecutar tira, hacemos que el caracter al que
hacía referencia ptr2 sea '\0'
        k = 1; //Le damos a k el
valor 1 para indicarle al hijo que imprima el pid.
```

```

                                child_pid = fork(); //Le
asignamos al proceso hijo el valor de fork(), para crearlo.
                                sleep(0);
                                }
                                else{ //Si no es asíncrono lo
ejecutamos como normalmente.
                                child_pid = fork(); //Le
asignamos al proceso hijo el valor de fork(), para crearlo.
                                waitpid(child_pid, &status, 0);
//Esperamos a que el proceso hijo finalicé, esto lo hacemos con waitpid, que
recibe como parámetros la etiqueta del hijo,
                                un entero que indica su estado, y 0
porque no usaremos opciones adicionales.
                                }

```

- **Ejecución proceso hijo:**

Tras recorrer todo nuestro programa prácticamente, entramos al hijo, cuya tiene el valor 0. Tenemos que comprobar si k es 1, en este caso tenemos que mostrar el pid del proceso hijo([1]getpid) con getpid(), y ejecutamos con execvp(tira[0], tira). La función execvp(unistd.h) recibe dos parámetros una string que es el comando principal(tira[0]) y el vector de strings con todos los elementos de la línea separados por espacios(tira). Para finalizar la ejecución del hijo usamos exit(0).

En caso de que la etiqueta del proceso hijo es menos uno, habrá ocurrido algún error al crear el proceso hijo.

```

if(child_pid == -1){
                                //si -1 ERROR AL CREAR EL HIJO
                                printf("Error al crar proceso hijo.");
                                return -1;
                                }

                                if(child_pid == 0){ //Si la etiqueta del proceso hijo es 0, significa
que ha sido llamado a través de la instrucción fork() y por lo tanto hay que
ejecutar las instrucciones.
                                if(k == 1){ //Si k es igual a 1 el usuario ha introducido
una operación asíncrona que debe ejecutarse de fondo.
                                printf("\n[+]%d\n.", getpid()); //Debemos
mostrar el pid del proceso hijo que se obtiene con getpid(), y que sigue la
estructura [+]pid. cuando se imprime en el shell
                                original.
                                }
                                execvp(tira[0], tira); //Execvp(de unistd.h) permite
ejecutar la instrucción que el usuario pase como parámetro, debiendo antes
trabajar un poco con la entrada del
                                usuario. Esta función recibe en primer lugar tira[0], el primer elemento
de la entrada del usuario, y el vector de strings tira que incluye todos los

```

elementos de la entrada del usuario, una vez son separados por espacios.  
exit(0); //Tras la ejecución del comando, finalizamos el hijo.

```
}  
  
}      }
```

- **Macro makefile:**

En el archivo Makefile definimos una secuencia de comando que permite crear un ejecutable de nuestro código con tan solo escribir make en la consola. Primero elimina los programas objetos en caso de que existan, en segundo lugar, crea un fichero objeto de funciones.c con gcc -c funciones.c. Después, realizamos lo mismo para Shell.c con gcc -c Shell.c que genera Shell.o. Por último, crea el ejecutable "shell" con el comando gcc -o Shell Shell.o funciones.o. Su código es el siguiente;

```
shell: shell.o funciones.o  
      gcc -o shell shell.o funciones.o
```

```
shell.o: shell.c  
      gcc -c shell.c
```

```
funciones.o: funciones.c  
      gcc -c funciones.c
```

```
clean:  
      rm *.o shell
```

- **Ejemplo de funcionamiento:**

```
codebin@codebin: ~/Documentos/Ejercicioshell
codebin@codebin:~/Documentos/Ejercicioshell$ ./shell
codebin@codebin:~/home/codebin/Documentos/Ejercicioshell$ cd ..
codebin@codebin:~/home/codebin/Documentos$ cd Ejercicioshell
codebin@codebin:~/home/codebin/Documentos/Ejercicioshell$ Echo Hola
Hola
codebin@codebin:~/home/codebin/Documentos/Ejercicioshell$ ls -l
total 52
-rw-rw-r-- 1 codebin codebin 3011 abr 22 18:24 funciones.c
-rw-rw-r-- 1 codebin codebin 1470 abr 22 18:24 funciones.h
-rw-rw-r-- 1 codebin codebin 2704 abr 22 20:37 funciones.o
-rw-rw-r-- 1 codebin codebin 163 abr 22 18:19 Makefile
-rwxrwxr-x 1 codebin codebin 17800 abr 22 20:37 shell
-rw-rw-r-- 1 codebin codebin 10537 abr 22 20:24 shell.c
-rw-rw-r-- 1 codebin codebin 3776 abr 22 20:37 shell.o
codebin@codebin:~/home/codebin/Documentos/Ejercicioshell$ ps&
[+]16534
codebin@codebin:~/home/codebin/Documentos/Ejercicioshell$      PID TTY      TIME C
MD
2450 pts/0    00:00:00 bash
16517 pts/0    00:00:00 shell
16534 pts/0    00:00:00 ps
exit
codebin@codebin:~/Documentos/Ejercicioshell$
```

```
codebin@codebin: ~/Documentos/Ejercicioshell
2450 pts/0    00:00:00 bash
16517 pts/0    00:00:00 shell
16534 pts/0    00:00:00 ps
exit
codebin@codebin:~/Documentos/Ejercicioshell$ cd ..
codebin@codebin:~/Documentos$ cd Ejercicioshell
codebin@codebin:~/Documentos/Ejercicioshell$ echo Hola
Hola
codebin@codebin:~/Documentos/Ejercicioshell$ ls -l
total 52
-rw-rw-r-- 1 codebin codebin 3011 abr 22 18:24 funciones.c
-rw-rw-r-- 1 codebin codebin 1470 abr 22 18:24 funciones.h
-rw-rw-r-- 1 codebin codebin 2704 abr 22 20:37 funciones.o
-rw-rw-r-- 1 codebin codebin 163 abr 22 18:19 Makefile
-rwxrwxr-x 1 codebin codebin 17800 abr 22 20:37 shell
-rw-rw-r-- 1 codebin codebin 10537 abr 22 20:24 shell.c
-rw-rw-r-- 1 codebin codebin 3776 abr 22 20:37 shell.o
codebin@codebin:~/Documentos/Ejercicioshell$ ps&
[1] 16686
codebin@codebin:~/Documentos/Ejercicioshell$      PID TTY      TIME CMD
2450 pts/0    00:00:00 bash
16686 pts/0    00:00:00 ps
```

En la imagen superior tenemos la ejecución de mi Shell y en la inferior replicamos los comandos en el propio Shell de Linux. Si nos fijamos el comportamiento es muy similar a diferencia de detalles como el directorio que en nuestro caso es la ruta completa y a que nuestro pid va acompañado de [+] en vez de [1].

- **Opinión:**

Me pasé demasiado tiempo intentando crear una función para separar la línea usuario,

algo que no conseguí, y también invertí mucho tiempo en como podía saber si la ejecución era asíncrona, lo conseguí tras realizar un gran número de intentos. En el resto del programa no tuve muchas dificultades.

- **Corrección de errores:**

El problema principal era que intentaba comprobar el valor de la etiqueta `child_pid` sin haberla creado, lo que impedía la ejecución. Desconozco porque en mi ordenador personal funcionaba, pero el caso es que en el de clase no lo hacía y era lo lógico. Tras esto desplacé la parte del proceso hijo al final del código, una vez se ha llamado a `fork()`, y puse la parte de la estructura del Shell y la lectura de comandos como parte del `main(padre)`. Además, observe que en mi programa anterior usaba una comparación `*ptr2 == "&"`, que daba un error por violación del segmento core. Lo corregí usando `tira[cont][longitud-1] == '&'`.