

SISTEMAS OPERATIVOS

Práctica 1: Microshell



1 Contenido

1	CONTENIDO	2
1.	OBJETIVOS Y PLANIFICACIÓN	4
2	SESIÓN 1: EL BUCLE PRINCIPAL	5
2.1	Objetivo	5
2.2	Descripción de las funciones a implementar	6
2.2.1	Función leerLinea()	6
2.2.2	Función visualizar ()	7
2.3	Pruebas a realizar	7
3	MÓDULO DE EJECUCIÓN	9
3.1	Objetivo	9
3.2	Descripción de las funciones a implementar	9
3.2.1	Función ejecutar(...)	9
4	MÓDULO DE REDIRECCIÓN	12
4.1	Objetivo	12
4.2	Descripción de las funciones a implementar	12
4.2.1	redireccion_ini()	13
4.2.2	Pipeline(...)	13
4.2.3	redirigir_entrada	14
4.2.4	redirigir_salida	14
4.2.5	cerrar_fd	15
4.3	Ejemplos guía	15
4.3.1	Orden simple sin redirección	15
4.3.2	Orden simple con redirección de salida	15
4.3.3	Orden simple con redirección de entrada	16
4.3.4	Orden compuesta sin redirección	16
4.3.5	Orden compuesta con redirecciones	17
5	AMPLIACIONES	19
5.1	Ejecución en background	19
5.2	Órdenes internas.	19
5.2.1	Orden cd	20

Sistemas Operativos

5.2.2	Orden exit	20
5.3	Señales	21
5.3.1	Introducción	21
5.3.1	Gestión de señales	21
6	ANEXO: MÓDULO ANALIZADOR	23

1. Objetivos y planificación

El objetivo de esta práctica es la programación de un *shell* o *intérprete de órdenes* simplificado que denominaremos “microshell”. Para llevar a cabo dicho objetivo será necesario trabajar con *llamadas al sistema* y en concreto con las responsables de gestión de procesos, redirecciones y tubos, y gestión de señales. Todas estas llamadas se estudian en las clases de teoría de la asignatura.

La práctica se compone de 4 sesiones de hora y media en las que se irán abordando de forma modular los distintos elementos que compone el “microshell”. El contenido de cada sesión se detalla en la siguiente tabla:

Semana 1	Realización del bucle principal
Semana 2	Módulo de ejecución de órdenes
Semana 3	Implementación de la redirección y tubos.
Semana 4	Ampliaciones y/o recuperación
Semana 5	Entrega y evaluación

Los archivos fuente necesarios estarán disponibles en el PoliformaT de la asignatura.

NOTA: Para poder comprobar el funcionamiento del shell completo, con independencia de la sesión en la que nos encontremos, se ofrecen todas las funciones implementadas con el sufijo `_profe`, es decir, `leerLinea_profe`, `visualizar_profe`, `redireccion_profe`, `ejecucion_profe` etc. La idea es ir sustituyéndolas por las implementadas por vosotros, en cada sesión

2 Sesión 1: El bucle principal

2.1 Objetivo

El **objetivo de la primera sesión** es desarrollar parte de la funcionalidad del **bucle principal**, en concreto, la relacionada con la adquisición de las órdenes introducidas por el usuario. Además, con el fin de comprobar que las órdenes introducidas se analizan correctamente, se debe implementar una función que permita visualizar por pantalla la información de éstas. La parte de análisis de la orden introducida se proporciona ya implementada. A modo de resumen:

Fichero de trabajo	ush.c
Funciones a implementar	leerLinea() visualizar()

Tabla 1 : Código de ush.c

```

// Declaraciones de funciones locales
void visualizar( void );
int leerLinea( char *linea, int tamanyLinea );

// Prog. ppal.
int main(int argc, char * argv[])
{
    char line[255];
    int res;
    char **m_ordenes;
    char ***m_argumentos;
    int *m_num_arg;
    int m_n;

    while(1){
        do{
            res=leerLinea_profe(line,MAXLINE);
            if (res== -2) {
                fprintf(stdout,"logout\n");
                exit(0);
            }
            if (res== -1){
                fprintf(stdout,"linea muy larga\n");
            }
        }while(res<=0);

        if (analizar(line)==OK){
            m_n=num_ordenes();
            m_num_arg=num_argumentos();
            m_ordenes=get_ordenes();
            m_argumentos=get_argumentos();
            if(m_n>0){
                if (pipeline_profe(m_n,fich_entrada(),
                                    fich_salida(),es_append(),es_background())==OK)
                    ejecutar_profe(m_n,m_num_arg,m_ordenes,m_argumentos,es_background());
            }
            visualizar_profe();
        }
        return 0;
    }
}

```

En la Tabla 1 aparece parte del fichero `ush.c`, en concreto la correspondiente a la declaración de las funciones a implementar (`leerLinea` y `visualizar`) y la función `main` que implementa el bucle principal.

Como se puede ver en el código, toda la funcionalidad del *shell* está ya implementada (funciones con el sufijo `_profe`), la idea es, en cada sesión, ir sustituyendo las funciones con sufijo por las vuestras. Pasemos a ver que hay que implementar exactamente en esta primera sesión.

2.2 Descripción de las funciones a implementar

2.2.1 Función `leerLinea(...)`

La función `leerLinea()` se debe implementar completamente y es la encargada de leer una línea desde teclado y mostrar el prompt:

```
int leerLinea(char *linea, int maxLinea);
```

Recibe el parámetro `linea`, que es un parámetro de salida. En concreto contiene el puntero a la cadena donde se debe guardar la línea leída en la función. El segundo parámetro es el tamaño máximo de caracteres que puede alcanzar dicha línea. El valor de retorno de la función es un entero que describe la condición de terminación de la función, de manera que:

Condición de finalización	Valor a retornar
Se ha leído correctamente la línea de teclado	Un valor <0, en concreto, el valor de caracteres leídos
Línea excesivamente larga (<maxLinea)	-1
Usuario introduce un Ctrl-D (EOF)	-2

Lo primero que debe hacer la función es mostrar el `prompt` en pantalla y quedar a la espera a que el usuario escriba una orden para poderla leer. El `prompt` que se desea visualizar estará compuesto por el directorio actual de trabajo, seguido de la constante `PROMPT` del módulo `defines.h`. Esta variable `PROMPT` se debe actualizar para que contenga el nombre del/de la estudiante. Para obtener el directorio actual de trabajo se puede utilizar la función `getcwd()`. Para conocer el funcionamiento de esta función, podéis utilizar las páginas del manual: `man getcwd`.

Una vez mostrado el `PROMPT`, debemos leer la línea de teclado, carácter a carácter e ir almacenándola en el parámetro `linea`. Para leer la orden se puede hacer uso de las funciones de lectura de carácter (por ejemplo, `getchar()`). La línea terminará cuando se lea de teclado un carácter '`\n`'. Es importante recordar que el carácter '`\n`' también debe ser almacenado en la variable `linea`. Finalmente hay que recordar que en esta función también se habrá de comprobar dos cosas más:

1. El tamaño de la línea no puede exceder el máximo número de caracteres que nos indican como segundo parámetro de la función (`maxLinea`). Si esto

pasara, se debe parar la lectura de caracteres y devolver un valor **-1**, para que el **main** pueda mostrar el correspondiente mensaje de error.

2. Si el usuario ha introducido la combinación de teclas CTRL-D (correspondiente al carácter final de fichero, EOF), esto debe provocar la finalización del Shell. Para ello la función ha de devolver un valor **-2** , y así, el **main**, mostrará el mensaje por pantalla de “ush: logout” y acabará su ejecución.

2.2.2 Función **visualizar()**

La función **visualizar()** va a permitir mostrar el resultado del análisis por pantalla.

```
void visualizar(void);
```

La función debe mostrar por pantalla toda la información sobre la línea de órdenes leída (y analizada):

- Número de órdenes y qué órdenes se han introducido
- Por cada por cada una de las órdenes, número de argumentos y el valor de los argumentos
- Si hay redirección de entrada, indicando el fichero de la redirección
- Si hay redirección de salida, indicando el fichero de la redirección y si se ha realizado una redirección en modo Append o en modo Trunk
- Si la orden se ha lanzado en FOREGROUND (primer plano) o en BACKGROUND (segundo plano)

Para poder mostrar toda la información se tendrá que hacer uso de las funciones de consulta del módulo analizador. Cómo mínimo se debe mostrar la información tal y como se hace en la función **visualizar_profe()**, cualquier mejora en la visualización se tendrá en cuenta para la nota de esta sesión.

2.3 Pruebas a realizar

Una vez realizados los pasos comentados en apartados anteriores, a continuación, se propone realizar una serie de pruebas para verificar que el analizador funciona correctamente y que todas las funciones del módulo analizador devuelven la información esperada.

En primer lugar, se propone hacer unas pruebas muy sencillas. Simplemente escribir órdenes simples y comprobar que nuestro shell nos contesta correctamente indicando que hay una única orden, el número de argumentos que contiene y cuales son. Estas órdenes podrían ser, por ejemplo:

```
$> ls -l
$> cp -i hola1 hola2
$> wc -l dkko/salida2
```

Además, indicará que no se redirige la entrada, ni la salida, y que la orden no se ejecuta en segundo plano (background).

Sistemas Operativos

Después, se propone realizar las pruebas anteriores de nuevo, pero lanzando las órdenes en segundo plano, para comprobar que efectivamente el analizador es capaz de detectarlo.

A continuación, se deben realizar unas pruebas para confirmar que el analizador gestiona correctamente la información relativa a redirección de la entrada y de la salida. Para ello podríamos teclear las siguientes órdenes:

```
$> ls -l > salida1
$> cat < entrada2
$> wc -l entrada1 >> salida2
$> cat < entrada3 >salida3
$> cat < entrada2 >> salida3
```

Nuestro shell nos debería indicar el nombre de los ficheros de redirección.

Por último, vamos a comprobar que es capaz de analizar correctamente órdenes compuestas por varias órdenes simples, diferenciando entre ellas y los argumentos correspondientes a cada una.

Algunos ejemplos que se podrían utilizar son los siguientes:

```
$> ps -ef | grep -A fich | wc -l
$> cat kko | grep -B fich | wc -w
$> ls -la | grep x | wc-l &
$> cat < entrada1 | grep fich | wc-l > salida1
```

3 Módulo de ejecución

3.1 Objetivo

En esta **segunda sesión** se va a desarrollar el **módulo de ejecución**, encargado de crear los procesos que ejecutarán las diferentes órdenes que puedan aparecer en una determinada línea de órdenes. A modo de resumen:

Fichero de trabajo	ejecucion.c
Función por implementar	ejecutar()

Tabla 2: Fichero ejecución.h

```
/*
 | E J E C U C I Ó N . c
+
| Version    :
| Autor      :
| Asignatura : SOP-GIROB
| Descripcion:
+
* /
```

```
#include "defines.h"
#include "redireccion.h"
#include "ejecucion.h"
#include <signal.h>

int ejecutar (int nordenes , int *nargs , char **ordenes , char ***args , int bgnd){

    // Código de creación de procesos

} // Fin de la función "ejecutar"
```

3.2 Descripción de la función a implementar

3.2.1 Función ejecutar(...)

Como se puede ver en la tabla la función ejecutar tiene 5 parámetros de entrada:

- **nordenes** → este parámetro indica el número de órdenes que hay en la línea de órdenes
- **nargs** → este parámetro, es un vector que indica, para cada orden, el número de argumentos que tiene
- **ordenes** → este parámetro es un vector de cadenas que indica la cadena de texto de cada una de las órdenes
- **args** → este parámetro es una matriz de cadenas que indica, para cada orden, los argumentos que tiene
- **bgnd** → este parámetro indica a verdadero que la orden se ha lanzado en background

Sistemas Operativos

Para entender mejor el contenido de los parámetros se muestra el siguiente ejemplo. Dada la orden

```
$> ls -l | grep -A 4 prof | sort -M
```

Parámetro	Valor
nordenes	3
nargs	nargs[0] = 2 //argumentos orden 0 nargs[1] = 4 //argumentos orden 1 nargs[2] = 2 //argumentos orden 2
ordenes	ordenes[0] = "ls" //orden 0 ordenes[1] = "grep" //orden 1 ordenes[2] = "sort" //orden 2
args	args[0][0] = "ls" args[0][1] = "-l" args[1][0] = "grep" args[1][1] = "-A" args[1][2] = "4" args[1][3] = "profe" args[2][0] = "sort" args[2][1] = "-M"
bgnd	False

Con esta información se deben crear los procesos necesarios para poder ejecutar la orden correctamente. Como ya se ha explicado en las clases de teoría, la creación de un nuevo proceso en UNIX siempre conlleva la utilización de ciertas llamadas al sistema. Esto podría resumirse en la siguiente secuencia:

1. El proceso padre (en este caso el shell) realiza una llamada fork() por cada proceso hijo a crear, de forma que, por cada orden de la línea de comandos, se cree un proceso hijo. La forma de generar los procesos hijos pueden seguir dos esquemas, como se puede ver en la Figura 1. Una vez creados todos los procesos necesarios, en el código de la función se debe diferenciar el código que ha de ejecutar el proceso padre (shell) del código del proceso hijo para que cada uno pueda hacer las acciones siguientes.

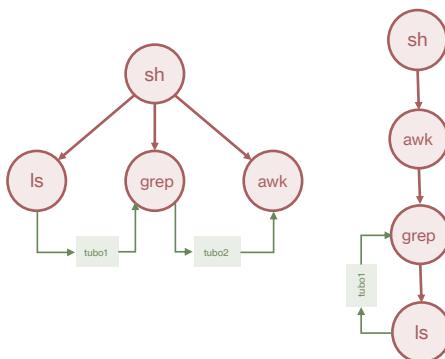


Figura 1.- Esquemas de generación de procesos

2. En el código de los **procesos hijo**, antes de cambiar su imagen por la del ejecutable

asociado a la orden que representan a través de la llamada execvp, se han de resolver las redirecciones de las órdenes. Como el módulo de redirección se implementará en la siguiente sesión, se pueden utilizar las funciones `redirigir_entrada_profe()`, `redirigir_salida_profe()` y `cerrar_fd_profe()` que ya están implementadas para poder comprobar el funcionamiento del módulo de ejecución en esta misma sesión.

3. Como se explicará en la siguiente sesión el **proceso padre**, antes de entrar en la espera de la finalización de los procesos hijos, ha de realizar la llamada a la función `cerrar_fd_profe()` del módulo de redirección. Para realizar la espera, el proceso padre lo puede hacer mediante las llamadas al sistema `wait()` o `waitpid()`. La forma de realizar tal espera depende de cómo se haya hecho la generación de los procesos hijos. Una vez el proceso padre salga de la espera debe retornar un OK.
4. En la versión sin ampliaciones no se va a implementar la opción de ejecución en segundo plano, por lo que el parámetro `bgnd`, no se debe de tener en cuenta.

4 Módulo de redirección

4.1 Objetivo

En esta **tercera sesión** se va a desarrollar el módulo de redirección, encargado de llevar a cabo las operaciones de redirección de los descriptores de E/S de los procesos que intervienen en la ejecución de una orden. A modo de resumen:

Fichero de trabajo	redireccion.c
Funciones a implementar	redireccion_ini()
	pipeline(...)
	redirigir_entrada(...)
	redirigir_salida(...)
	cerrar_fd()

Tabla 3: Fichero redireccion.c

4.2 Descripción de las funciones a implementar

En este fichero se declaran las funciones que se definen en el módulo `redirección.c` y que tendrán que ser implementadas en esta parte de la práctica.

Para entender mejor las tareas a realizar en el módulo, primero vamos a describir la estructura de datos en la que se sustenta todo él. Esta estructura se llama `red_ordenes` está declarada en el fichero `defines.h`. Dicha estructura contiene los descriptores de fichero necesarios para manejar las redirecciones en la línea de órdenes y se define de la siguiente forma:

```
struct {
    int entrada;
    int salida;
}REDIRECCION_ORDENES[PIPELINE]
```

En la estructura `red_ordenes`, cada componente representa el valor de las entradas 0 y 1 de la tabla de descriptores abiertos de un proceso de la línea de órdenes. En esta estructura nos anotamos todas las redirecciones a realizar, para una vez creados los procesos, poderlas hacer.

4.2.1 redireccion_ini()

La función `redireccion_ini` se encarga de inicializar de la estructura `red_ordenes` a los valores por defecto.

La sintaxis de la función `redireccion_ini` es la siguiente:

```
extern int redireccion_ini(void);
```

El trabajo que debe realizar la función consiste en asignar los descriptores por defecto a los componentes `entrada` y `salida` de la estructura `red_ordenes` para cada orden que forme parte de la línea de órdenes.

4.2.2 Pipeline(...)

Esta función se encarga de crear los tubos necesarios ($n-1$) para ejecutar n órdenes en tubería y redirigir la entrada y salida estándar de un proceso. La sintaxis de la función `pipeline` es la siguiente:

```
extern int pipeline(int nordenes,
                    char *infile,
                    char *outfile,
                    int append,
                    int bgnd);
```

Los argumentos de entrada de esta función consisten en:

- `nordenes`: valor entero que representa el número de comandos de la orden.
- `infile`: cadena que representa el fichero de redirección de la entrada estándar, si la hay. Contiene "" en caso de no haber redirección de entrada
- `outfile`: cadena que representa el fichero de redirección de la salida estándar, si

la hay. Contiene “” en caso de no haber redirección de salida.

- **append:** entero (1 ó 0) que indica si la redirección de la salida se añade al final del fichero utilizado (modo append) o debe sobrescribir éste (modo trunk).
- **bgnd:** entero (1 ó 0) que representa la ejecución en modo “background” o no.

La función **pipeline** se encarga de actualizar las componentes de un vector que mantiene todos los descriptores de fichero necesarios para “construir” las redirecciones y tuberías necesarias para esta línea de órdenes. Dicho vector está representado por la variable **red_ordenes** .

El desarrollo de la función **pipeline** requiere una serie de pasos que consisten en:

1. Realizar la llamada a la función **redireccion_ini** para inicializar el vector **red_ordenes**.
2. Implementar el código que se encargue de preparar la redirección de la entrada y salida estándar, si fuera necesario. Para ello, se almacenará la información sobre los descriptores de entrada y/o salida redirigidos, en el vector **red_ordenes**.
3. Crear tantos tubos como sean necesarios mediante la llamada **pipe**, para conectar las órdenes recibidas y guardar los descriptores devueltos por cada llamada en el vector **red_ordenes**.
4. En caso de ejecución en segundo plano se redirigirá la entrada estándar de la primera orden al dispositivo **/dev/null**.

La función **pipeline** devuelve un cero en caso de ERROR y un 1 si es correcta su ejecución (OK).

4.2.3 **redirigir_entrada**

Se trata de la función que, en caso de ser necesario, redirige la entrada estándar de un proceso al descriptor del tubo o fichero correspondiente. Su sintaxis consiste en:

```
extern int redirigir_entrada(int i);
```

donde el entero **i** corresponde al número de orden dentro de la línea de órdenes introducida (empieza a numerar desde 0). La función **redirigir_entrada** ha de consultar y utilizar el contenido de la estructura **red_ordenes** para determinar si hay redirección. La función devuelve un cero en caso de ERROR y un 1 si es correcta su ejecución (OK).

4.2.4 **redirigir_salida**

Se trata de la función que redirige la salida estándar de un proceso al descriptor del tubo o fichero correspondiente. Su sintaxis consiste en:

```
extern int redirigir_salida(int i);
```

donde el entero *i* corresponde al número de orden dentro de la línea de órdenes introducida (empieza a numerar desde 0). La función `redirigir_salida` ha de consultar y utilizar el contenido de la estructura `red_ordenes` para determinar si hay redirección. La función devuelve un cero en caso de ERROR y un 1 si es correcta su ejecución (OK).

4.2.5 cerrar_fd

La sintaxis de la función `cerrar_fd` consiste en:

```
extern int cerrar_fd(void);
```

Se encarga de cerrar los descriptores creados y que ya han sido utilizados en la redirección de la entrada o salida estándar de un proceso o en la generación de tubos. Hay que tener en cuenta que mientras no se cierren todos los descriptores de escritura sobre el tubo, el lector del tubo no recibirá la marca de final de fichero (EOF) y permanecerá bloqueado, esperando más datos. Los descriptores estándar deben permanecer abiertos (0, 1 y 2)

4.3 Ejemplos guía

Con el fin de saber cuál es el resultado esperado de las funciones anteriores, vamos a describir una serie de ejemplos

4.3.1 Orden simple sin redirección

Como primer ejemplo vemos el contenido de la estructura de datos `red_ordenes` y de las tablas de descriptores de los diferentes procesos implicados:

```
$> ls -l
```

Estructura de <code>red_ordenes</code>	<code>red_ordenes[0].entrada = 0</code> <code>red_ordenes[0].salida = 1</code>
Tabla de descriptores del Shell antes de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR
Tabla de descriptores del Shell después de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR
Tabla de descriptores del proceso que ejecuta la orden antes del exec	0 SDTIN 1 STDOUT 2 STDERR

4.3.2 Orden simple con redirección de salida

Si plantamos ahora una orden con redirección de salida:

```
$> ls -l > sal
```

Estructura de red_ordenes	red_ordenes[0].entrada = 0 red_ordenes[0].salida = 3
Tabla de descriptores del Shell antes de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR 3 sal
Tabla de descriptores del Shell después de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR
Tabla de descriptores del proceso que ejecuta la orden antes del exec	0 SDTIN 1 sal 2 STDERR

4.3.3 Orden simple con redirección de entrada

Si la redirección es de entrada:

```
$> cat < ent
```

Estructura de red_ordenes	red_ordenes[0].entrada = 3 red_ordenes[0].salida = 1
Tabla de descriptores del Shell antes de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR 3 ent
Tabla de descriptores del Shell después de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR
Tabla de descriptores del proceso que ejecuta la orden antes del exec	0 ent 1 STDOUT 2 STDERR

4.3.4 Orden compuesta sin redirección

Si en vez de una orden simple planteamos dos órdenes conectadas con una tubería:

```
$> ls -l | grep txt
```

Estructura de red_ordenes	red_ordenes[0].entrada = 0 red_ordenes[0].salida = 4 red_ordenes[1].entrada = 3 red_ordenes[1].salida = 1
Tabla de descriptores del Shell antes de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR 3 Tubo_lectura 4 Tubo_escritura
Tabla de descriptores del Shell después de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR
Tabla de descriptores de los procesos que ejecuta la orden antes del exec	<u>Proceso ls</u> 0 STDIN 1 Tubo_escritura 2 STDERR <u>Proceso grep</u> 0 Tubo_lectura 1 STDOUT 2 STDERR

4.3.5 Orden compuesta con redirecciones

Si en vez de una orden simple planteamos dos órdenes conectadas con una tubería:

```
$> cat <lista | grep Perez > nPerez
```

Estructura de red_ordenes	red_ordenes[0].entrada = 5 red_ordenes[0].salida = 4 red_ordenes[1].entrada = 3 red_ordenes[1].salida = 6
Tabla de descriptores del Shell antes de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR 3 Tubo_lectura 4 Tubo_escritura 5 lista 6 nPerez

Sistemas Operativos

Tabla de descriptores del Shell después de llamar a ejecutar	0 SDTIN 1 STDOUT 2 STDERR						
Tabla de descriptores de los procesos que ejecuta la orden antes del exec	<p><u>Proceso cat</u></p> <table border="1"><tr><td>0 lista</td></tr><tr><td>1 Tubo_escritura</td></tr><tr><td>2 STDERR</td></tr></table> <p><u>Proceso grep</u></p> <table border="1"><tr><td>0 Tubo_lectura</td></tr><tr><td>1 nPerez</td></tr><tr><td>2 STDERR</td></tr></table>	0 lista	1 Tubo_escritura	2 STDERR	0 Tubo_lectura	1 nPerez	2 STDERR
0 lista							
1 Tubo_escritura							
2 STDERR							
0 Tubo_lectura							
1 nPerez							
2 STDERR							

5 Ampliaciones

En esta última sesión se tendrá que completar el microshell. Si el alumno ya ha terminado su implementación, se puede optar a mejorar la calificación de esta práctica implantando algunas ampliaciones. En principio se proponen dos:

- 1) Implementar las órdenes en background
- 2) Implantar alguna orden interna. Por ejemplo: cd y exit.
- 3) Implantar la gestión de señales

5.1 Ejecución en background

La primera de las posibles ampliaciones se trata de introducir la posibilidad en nuestro shell de ejecutar órdenes en segundo plano, lanzadas mediante el operador correspondiente ("&").

La solución más elegante sería hacerlo de una forma similar a la empleada por el bash o el ksh. En estos intérpretes, cuando se ha detectado que la orden en segundo plano ha terminado, se muestra por pantalla una línea informando del tipo de terminación que ha tenido la orden (si es normal, se muestra la palabra "Done" por pantalla; si ha sido terminada con una señal, se muestra la palabra "Killed"; si ha terminado con error, aparecerá la palabra "Exit" seguida por el código de error devuelto por el programa, ...), y a continuación se escribe también la línea de órdenes que haya producido tales resultados. Esta solución se considerará una ampliación extra y, en caso de realizarla, mejoraría la calificación obtenida. No obstante, lo que se exigirá como mínimo en esta ampliación será una gestión que no produzca como resultado procesos zombies a partir de las órdenes lanzadas en segundo plano.

5.2 Órdenes internas.

Reciben el nombre de órdenes internas todas aquellas que no precisen la ejecución de ningún programa externo para desarrollar sus funciones. Esto implica que el código de tales órdenes internas debe encontrarse en el propio intérprete.

La utilización de órdenes internas implicará ciertos cambios en el bucle principal de nuestro intérprete de órdenes. En primer lugar, habrá que guardar en algún vector todos los nombres de las órdenes internas que nuestro shell sea capaz de reconocer y ejecutar. Así, cuando el usuario introduzca una nueva línea deberá comprobarse antes de iniciar su ejecución que cada uno de los nombres de orden facilitados en esa línea no corresponda con alguna de estas órdenes internas. Si alguna de las órdenes fuera interna, ésta debe ser procesada directamente y no debe iniciarse la creación de un nuevo proceso para ella, ni usar la llamada exec() para ejecutar el programa correspondiente, ya que en este caso no existiría.

En esta sesión únicamente se pide implantar dos órdenes internas: cd y exit. La primera se utiliza para cambiar el directorio de trabajo actual del intérprete de órdenes, mientras que la segunda sirve para terminar la sesión de trabajo actual, forzando a que termine la ejecución del intérprete.

Veamos en más detalle cada una de estas órdenes.

5.2.1 Orden cd

Esta orden admite tres tipos de argumentos:

- a) La ruta absoluta o relativa del directorio al que se pretende pasar. Éste es el uso más común. Para servir estas peticiones basta con utilizar el nombre pasado como argumento como parámetro de una llamada al sistema `chdir()`.
- b) Sin argumentos. En este caso, el intérprete pasa a utilizar como directorio de trabajo actual el que tiene asignado originalmente el usuario. Para ello utiliza la variable de entorno `HOME`. Es decir, se emplea de nuevo la llamada al sistema `chdir()`, pero ahora se utiliza como parámetro para esta llamada el valor de la variable de entorno antes citada. El vector que contiene todas las variables de entorno facilitadas a un programa puede obtenerse como tercer argumento de la función `main()`.

La solución más sencilla para conocer el valor de la variable de entorno `$HOME` consiste en utilizar la función `POSIX getenv()` (véase su página de manual), facilitando como único argumento el nombre de la variable a buscar. Al hacerlo así, esta función nos devolvería como resultado un puntero a una cadena que contendría el valor de la variable pedida, o el valor `NULL` en caso de no haber podido encontrar tal variable en el entorno facilitado al proceso. La cabecera de esta función es la siguiente:

```
#include <stdlib.h>
char *getenv(const char *nombre);
```

- c) Por último, también se admite la utilización del argumento “-“ (guión), mediante el que resulta posible especificar que se quiere pasar al directorio en el que nos encontrábamos anteriormente. Para implantar esto, basta con recordar cuál fue el último directorio en el que nos encontrábamos y, si se ha utilizado un “cd -“, entonces se vuelve a él. Nótese que no es necesario guardar una pila de directorios, sino sólo el utilizado previamente. Es decir, que, si utilizáramos dos veces seguidas esta variante del “cd”, volveríamos al directorio en el que nos encontrábamos antes de haber utilizado esas dos órdenes. Para implantar esta variante, deberá utilizarse también la llamada al sistema `getcwd()`.

No se exige la implantación de estas tres variantes, sino sólo la primera de ellas. En caso de implantar cualquiera de las dos restantes, la nota se vería mejorada proporcionalmente.

5.2.2 Orden exit

Esta orden finaliza la ejecución del intérprete. Para ello utiliza la llamada al sistema `exit()`. Es posible facilitar un argumento numérico, que también sería usado como parámetro para realizar la llamada al sistema. En caso de no utilizar argumentos, se utilizará el valor cero como parámetro de la llamada.

5.3 Señales

5.3.1 Introducción

El tratamiento de señales en los intérpretes de órdenes UNIX tradicionales ha dependido en gran medida del tipo de gestión de procesos que éstos soportaban. Los intérpretes más complejos proporcionaban soporte para la gestión de grupos de procesos y utilizaban éstos para modelar el concepto de “trabajo” y permitir la parada, el cambio de primer a segundo plano, o de segundo a primer plano de estos “trabajos”. En estos casos se precisaba gestionar algunas señales adicionales que tenían cierta influencia en la gestión de los trabajos, así como la creación de los grupos de procesos necesarios para soportarlos. Por ejemplo, el intérprete bash ignora por omisión las señales SIGTERM, SIGQUIT y SIGINT (esta última no es realmente ignorada, sino que tiene un manejador asociado para que la orden interna `wait` sea interrumpible). Cuando su control de trabajos está habilitado, también pasa a ignorar las señales SIGTTIN, SIGTTOU y SIGTSTP. Para todos los procesos lanzados en primer plano, estas señales no estarán ignoradas, pero para los lanzados en segundo plano, SIGINT y SIGQUIT sí que son ignoradas si el control de trabajos está inhabilitado. Además de esto, el tratamiento por omisión realizado por el intérprete puede modificarse mediante la orden interna `trap`, al menos en lo que hace referencia al tratamiento de las señales dirigidas al propio intérprete. Como puede verse, bash no utiliza una gestión sencilla.

El estándar POSIX no requiere que cualquier intérprete implemente esta gestión avanzada, por lo que únicamente exige un tratamiento de las señales dirigido a evitar que el propio intérprete pueda ser eliminado al utilizar ciertas combinaciones de teclas.

En esta sesión deberá ampliarse el módulo de ejecución, así como el subprograma principal del intérprete para dar soporte a este tratamiento de señales. Para ello habrá que utilizar debidamente la llamada al sistema `sigaction()`, con la que se podrá modificar la gestión de señales que llevará a cabo el shell o los procesos que éste genere, dependiendo de dónde se use. Veamos entonces qué tipo de gestión se tendrá que implantar en esta sesión de prácticas.

5.3.1 Gestión de señales

En nuestro intérprete de órdenes no vamos a realizar ningún tipo de gestión de trabajos, por lo que la **gestión de las señales será sencilla**. Los requerimientos para ésta se centrarán en:

- El **intérprete deberá ignorar** cualquiera de las siguientes señales:
 - **SIGINT**, o señal de interrupción de teclado. Se puede generar normalmente mediante la combinación de teclas [Ctrl]+[C]. Su tratamiento por omisión es terminar al proceso que la reciba. En los intérpretes con gestión de trabajos, esta señal sólo se dirige al grupo de procesos en primer plano, pues los generados en segundo plano utilizan otros grupos de procesos que no tienen terminal asociada y que, por tanto, no pueden recibir ninguna señal generada desde teclado. En nuestro intérprete no vamos a preocuparnos por la gestión de trabajos.
 - **SIGQUIT**, o señal de terminación por teclado. Se genera con la combinación de teclas [Ctrl]+[\] (aunque tanto la combinación para ésta como para la anterior señal pueden modificarse con la orden `stty`. De hecho, la

combinación en uso puede consultarse con un “`stty -a`”). Su tratamiento por omisión es terminar al proceso que la reciba, realizando además un volcado de memoria en un fichero (al que se le suele dar como nombre “`core`”). Al igual que en el caso anterior, esta señal únicamente podrá ser entregada a los procesos que tengan acceso a la terminal donde se haya pulsado la combinación de teclas correspondiente.

- **SIGTTIN.** Esta señal es enviada por el sistema operativo cuando un proceso sin acceso a la terminal intenta efectuar una lectura de teclado. Como ya hemos comentado anteriormente, los procesos en segundo plano no deberían tener acceso a la terminal. Por tanto, lo que tratamos de evitar al ignorar esta señal es que nuestro intérprete quede parado (ya que pararse es el tratamiento por omisión para esta señal) cuando sea lanzado desde la línea de órdenes en segundo plano.
- **SIGTTOU.** Esta señal es muy similar al anterior, pero relacionada con la salida de la terminal (es decir, con el acceso a la pantalla) en lugar de la entrada.
- Por su parte, los procesos lanzados en segundo plano sólo tendrán que ignorar las señales `SIGINT` y `SIGQUIT`. Esto es así porque no vamos a realizar ninguna gestión de sesiones ni grupos de procesos y, en nuestro intérprete de órdenes, siempre tendremos a todos los procesos asociados con la terminal. En un intérprete más avanzado (cualquiera de los intérpretes “estándares” de UNIX: `ksh`, `csh` o `bash`, por ejemplo), al realizar gestión de grupos, los procesos en segundo plano no estarían asociados a la terminal y jamás podrían recibir tales señales. Por otra parte, nuestra gestión simplificada tendrá la ventaja de que los procesos en segundo plano que se generen con nuestro shell jamás podrán recibir las señales `SIGTTIN` ni `SIGTTOU`, por lo que no merecerá la pena realizar ningún tratamiento para ellas. No obstante, hay que tener especial cuidado con la entrada de los procesos en segundo plano. En el módulo de redirección habrá tenido que comprobarse que la entrada de estas órdenes esté redirigida y, si no lo estaba, efectuar una redirección utilizando el fichero “`/dev/null`” con lo que la orden en segundo plano leería de inmediato un fin de fichero. Si no se hizo así, nos encontraremos con el problema de tener a varios procesos compitiendo por la entrada que quiera facilitar el usuario, con resultados bastante caóticos.
- Por último, los **procesos en primer plano** deberían tener el **tratamiento por omisión** para todas las señales. Por tanto, habrá que recuperar tal tratamiento utilizando las llamadas al sistema apropiadas, pues estos procesos habrán heredado la gestión realizada por nuestro intérprete.

En esta sesión tendremos que añadir el código necesario para implantar esta gestión dentro nuestros módulos. Para ello habrá que **modificar** el fichero `ush.c` para tener en él la gestión relacionada con el propio intérprete y tras esto, modificar también el fichero `ejecucion.c`, donde se incluirá la gestión tanto para los procesos en primer plano como para los procesos en segundo plano, si es que se ha implementado esa ampliación.

Todas estas modificaciones tendrán que utilizar las llamadas y estructuras de datos relacionadas con señales. Para ello es conveniente repasar la documentación existente sobre la llamada al sistema `sigaction()` (tanto los apuntes de clase como las páginas de manual).

6 Anexo: Módulo Analizador

A continuación se ofrece una descripción de las funciones del módulo Analizador.c:

Función	Descripción
Módulo analizador	
void analizador_ini(void);	Inicializa el analizador
int analizar(char *s);	Analiza una orden según la sintaxis descrita. Retorna OK si es sintácticamente correcto o ERROR si hay error.
int num_ordenes(void);	Número de ordenes analizadas en la orden analizar
char **get_ordenes(void);	Devuelve un vector con las ordenes analizadas en la orden analizar
int *num_argumentos(void);	Devuelve un vector de enteros con el número de argumento de cada orden
char ***get_argumentos(void);	Matriz de cadenas con los argumentos
char *fich_salida (void);	Devuelve el fichero de salida
Char *fich_entrada (void);	Devuelve el fichero de entrada
int es_append(void);	Indica si la salida debe ser añadida al fichero (>)
int es_background(void);	Indica si las ordenes analizadas se han lanzado en background (&)
char *error_sintactico(void);	Devuelve una descripción del error sintáctico producido al analizar
Módulo Redirección	
void redireccion_ini(void);	Inicializa el módulo de redirección
int pipeline(int ncmd, char * infile, char * outfile, int append, int bgnd);	Crea los tubos necesarios (n-1) para ejecutar n ordenes en tubería y redirige la STD_INPUT y STD_OUTPUT de un proceso. Debe especificarse redirecciones de entrada, de salida y background. Parámetros: <ul style="list-style-type: none">• ncmd: número de órdenes del <i>pipe</i>.• infile: fichero donde se redirecciona la STD_INPUT.• outfile: fichero donde se redirecciona la STD_OUTPUT.• append: (TRUE o FALSE) indica si el fichero al que se redirecciona la salida standard se añade la salida al final o debe ser sobreescrito.• bgnd: (TRUE o FALSE) indica si la instrucción se ejecuta en segundo plano (<i>modo background</i>).
int redirigir_entrada(int i);	Redirige STD_INPUT del proceso i al descriptor fd del tubo/fichero correspondiente de la tubería
int redirigir_salida(int i);	Redirige la STD_OUTPUT del proceso i al descriptor fd del tubo/fichero correspondiente de la tubería.
cerrar_fd(void);	Cierra los descriptores fd de los tubos creados con <i>pipeline</i> .
Módulo Ejecución	
int ejecutar (int nordenes, int *nargs, char **ordenes, char ***args, int bgnd);	Crea los procesos para ejecutar el argumento especificado en "ordenes". Parámetros <ul style="list-style-type: none">• nordenes: número de órdenes del <i>pipe</i> que se quiere ejecutar.• nargs: vector con el número de argumentos de cada orden.• ordenes: vector de cadenas con el nombre de las órdenes.• args: matriz de cadenas con los argumentos de cada orden, es decir, en la posición (i,0) se encuentra la orden i-ésima, y en la posición (i,n) con n>0 se encuentra el argumento n de la posición i-ésima.• bgnd: (TRUE o FALSE) indica si la instrucción se ejecuta en modo background. Retorna OK si ha ejecutado correctamente.