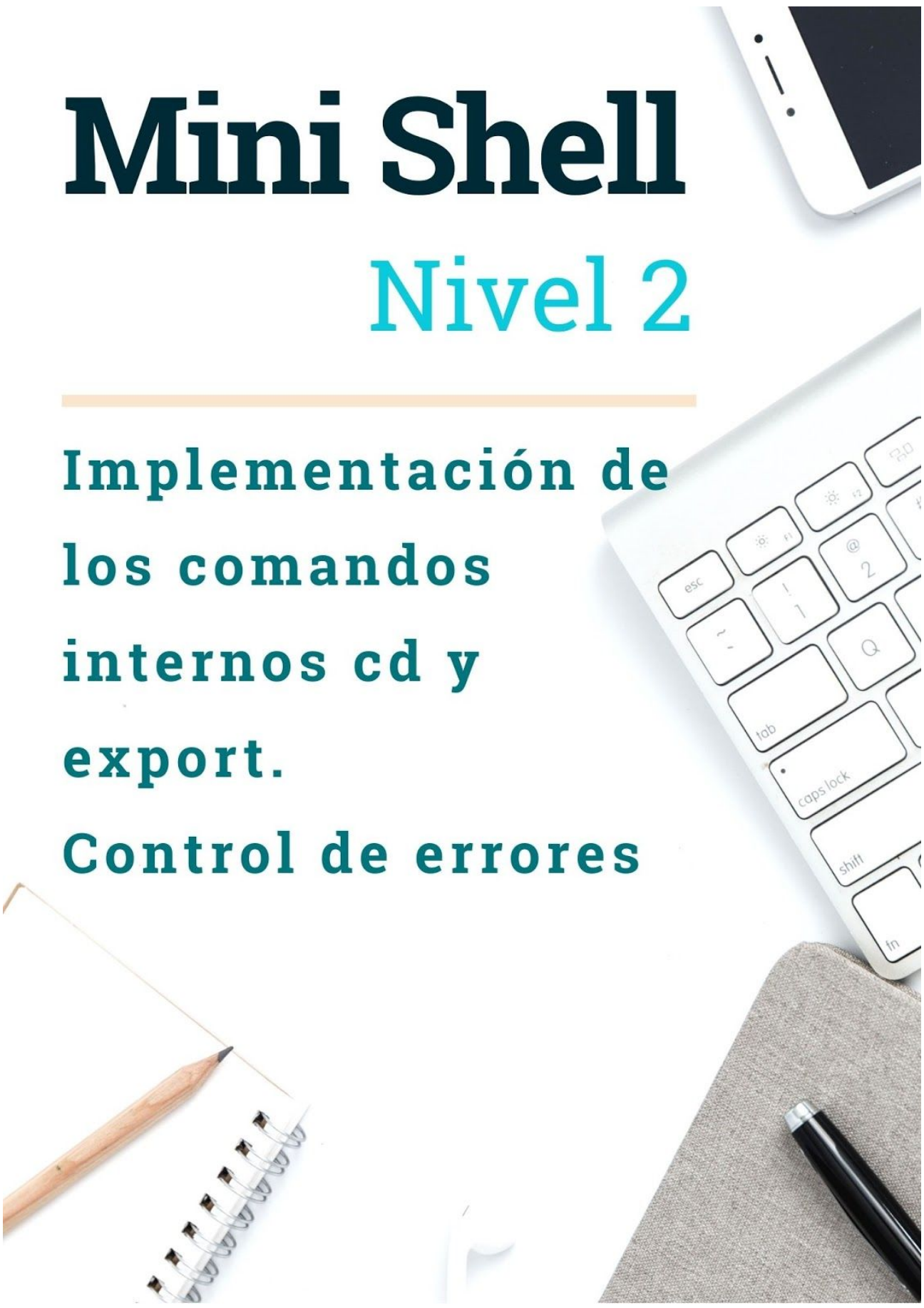
Mini Shell

Nivel 2

Implementación de los comandos internos cd y export.
Control de errores



Indice

Mini Shell
Nivel 2

Implementación de los comandos internos cd y export. Control de errores.

En este nivel implementaremos los comandos internos **cd** y **export** de nuestro mini shell, y también veremos cómo tratar los **errores** que pueden producir las **llamadas a sistema**, y en general cómo utilizar el flujo estándar de errores para mostrar cualquier error de nuestros programas.

¿Cuáles son los retos de este nivel?

1. Gestionar los errores de las llamadas al sistema
2. Implementar los comandos internos **cd** (para cambiar de directorio) y **export** (para asignar un valor a una variable de entorno)

¿Qué conceptos trabajamos con estos retos?

- Control de errores de las llamadas al sistema

- Cambio de directorio con la llamada al sistema **chdir**()
- Obtención del directorio de trabajo mediante la llamada al sistema **getcwd**(). Ejemplo:

```c
#include <unistd.h>
#include <stdio.h>
#include <limits.h>
#define COMMAND_LINE_SIZE 1024

int main() {
    char cwd[COMMAND_LINE_SIZE];
     if (getcwd(cwd, COMMAND_LINE_SIZE]) != NULL) {
      printf("Directorio actual: %s\n", cwd);
  } else {
      perror("getcwd() error");
      return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

- Asignación de un valor a una variable de entorno con **setenv**()

Implementación de los comandos internos cd y export. Control de errores.

¿Cuáles son los detalles de implementación?

Hay que ampliar las siguientes funciones:

int internal_cd(char **args);
Utiliza la llamada al sistema **chdir**() para cambiar de directorio[1].
En este nivel, a modo de test, muestra por pantalla el directorio al que nos hemos trasladado. Para ello usa la llamada al sistema **getcwd**() (en niveles posteriores eliminarlo).

Si queréis que se os actualice el prompt al cambiar de directorio, podéis cambiar el valor de la variable de entorno PWD mediante **setenv**() y utilizarla después en la función imprimir_prompt(), aunque también podéis usar **getcwd**() en vez de PWD para imprimir el prompt.

El comando "cd" sin argumentos ha de enviar al valor de la variable HOME.

Adicionalmente se puede implementar el **cd avanzado**[2]. En ese caso en la sintaxis tendremos que admitir más de 2 elementos. Podéis emplear la función **strchr**() para determinar si el token guardado en args[1] contiene comillas simples o dobles, o el caràcter \.

int internal_export(char **args);
Descompone en tokens el argumento NOMBRE=VALOR (almacenado en args[1]), por un lado el nombre y por otro el valor.
Notifica de la sintaxis correcta si los argumentos no son los adecuados, utilizando la salida estándar de errores *stderr*.
En este nivel, muestra por pantalla mediante la función **getenv**() el **valor inicial** de la variable (en niveles posteriores eliminarlo).
Utiliza la función **setenv**() para asignar el nuevo valor.
En este nivel, muestra por pantalla el **nuevo valor** mediante la función **getenv**() para comprobar su funcionamiento (en niveles posteriores eliminarlo).

[1] Si en el prompt usáis la variable de entorno PWD, tendréis que actualizarla previamente con setenv(). Para que los cambios fueran permanentes se debería editar el archivo **/home/user/.bashrc**.
[2] Bash permite acceder a **directorios cuyos nombres contienen espacios** poniendo todo el nombre entre comillas (dobles o simples) o bien poniendo el carácter escape antes del espacio en blanco.
Ejemplo para acceder a un directorio denominado “sistemas operativos”:
$ cd “sistemas operativos”
$ cd 'sistemas operativos'
$ cd sistemas\ operativos

Observaciones:

- Ahora que ya sabréis mostrar los valores de las variables de entorno, podéis utilizar algunas como parte del prompt, si no lo habéis hecho ya en el nivel anterior.

¿Cómo comprobamos que hemos superado este nivel?

- Con vuestras propias pruebas, mediante los mensajes por pantalla provisionales, que muestran los cambios de directorio y los nuevos valores de las variables de entorno.
- Y también podéis ejecutar vuestro programa **nivel2.c**[3] e ir escribiendo en la línea de comandos los ejemplos que se exponen a continuación, comparando vuestros resultados con los mostrados aquí (teniendo en cuenta cuál es vuestro USER y vuestro directorio raíz como usuarios).

```
uib:$ ./nivel2
$ cd
[internal_cd()→ /home/uib]
$ cd practicasSO/SOI/miP2/niveles [4]
[internal_cd()→ /home/uib/practicasSO/SOI/miP2/niveles]
$ cd nivel8
chdir: No such file or directory
$ cd ..
[internal_cd()→ /home/uib/practicasSO/SOI/miP2]
$ cd niveles
[internal_cd()→ /home/uib/practicasSO/SOI/miP2/niveles]
$ cd ../../../../Documentos
[internal_cd()→ /home/uib/Documentos]
$ cd
[internal_cd()→ /home/uib]
$ cd practicasSO/SOI/"prueba dir" [5]
[internal_cd()→ /home/uib/practicasSO/SOI/prueba dir]
$ cd ..
[internal_cd()→ /home/uib/practicasSO/SOI]
$ cd practicasSO/SOI/'prueba dir'
[internal_cd()→ /home/uib/practicasSO/SOI/prueba dir]
$ cd ..
```

[3] nivel2.c lo creáis a partir de nivel1.c copiando el código que ya teníais allí. Hay que conservar todos los niveles por separado.
[4] Para poder trasladarse a ese directorio tiene que estar previamente creada la ruta (comando mkdir)
[5] Adaptar la ruta a una vuestra donde tengáis un directorio que contenga un espacio en el nombre

Implementación de los comandos internos cd y export. Control de errores.

```
[internal_cd()→ /home/uib/practicasSO/SOI]
$ cd practicasSO/SOI/prueba\ dir
[internal_cd()→ /home/uib/practicasSO/SOI/prueba dir]
$ pwd
$ export USER=Yo
[internal_export()→ nombre: USER]
[internal_export()→ valor: Yo]
[internal_export()→ antiguo valor para USER: uib]
[internal_export()→ nuevo valor para USER: Yo]
$ export USER=======Yodenuevo
[internal_export()→ nombre: USER]
[internal_export()→ valor: ======Yodenuevo]
[internal_export()→ antiguo valor para USER: Yo]
[internal_export()→ nuevo valor para USER: ======Yodenuevo]
$ export a
[internal_export()→ nombre: a]
[internal_export()→ valor: (null)]
Error de sintaxis. Uso: export Nombre=Valor
$ export
Error de sintaxis. Uso: export Nombre=Valor
$ exit
uib:$
```

Anexo 1: Control de errores de las llamadas al sistema

Los programas deben siempre comprobar después de una **llamada al sistema**[6] si todo es correcto. Cuando se produce un error al realizar una llamada al sistema, ésta devuelve el valor -1. Todo proceso contiene una variable global llamada ***errno***, de manera que cuando una llamada al sistema termina de forma satisfactoria, el valor de esta variable permanece inalterado, pero cuando la llamada falla, en *errno* se almacena un código de error.
errno no da información detallada sobre el error que se ha producido pero el lenguaje de programación C proporciona las funciones **perror ()** y **strerror ()** que se pueden utilizar para mostrar el mensaje de texto asociado a la variable *errno* .

- perror() muestra la cadena que se pasa a la función, seguida de dos puntos, un espacio, y luego la representación textual del valor de la variable errno actual.
- strerror() devuelve un puntero a la representación textual del valor actual de la variable *errno*.

Ejemplo:

```c
// uso_perror.c - Funciones perror() y fprintf() con stderr y errno

#include <stdio.h>
#include <sys/file.h>
#include <errno.h> //errno
#include <string.h> //strerror()

int main()
{
  int fd;
  fd=open("/asdf", O_RDONLY); //No existe el fichero
  if (fd==-1) {
     fprintf(stderr, "Error %d: %s\n", errno, strerror(errno));  //o bien:
     perror("Error");
  }
  if ((fd=open("/", O_WRONLY))==-1) {
     fprintf(stderr, "Error %d: %s\n", errno, strerror(errno)); //o bien:
     perror("Error");
  }
}
```

[6] Las llamadas al sistema están recopiladas en la sección 2 de Man. Ejemplos en nuestra aventura: open(), close(), chdir(), getcwd(), dup(), dup2(), execvp(), exit(), fork(), getpid(), kill(), pause(), signal(), wait(), waitpid(). Hay algunas llamadas al sistema que siempre tienen éxito, y no retornan ningún valor para indicar si se ha producido un error; ejemplos en nuestra aventura: getpid(), getppid()

Cuando el código se compila y ejecuta, se produce el siguiente resultado:

```
uib ~/Documentos/Asignaturas/21708-SOI/practicas/ejemplos$ ./uso_perror
Error 2: No such file or directory
Error: No such file or directory
Error 21: Is a directory
Error: Is a directory
```

En general para mostrar errores es preferible utilizar la salida de error estándar ***stderr*** (con **fprintf**()), en vez de utilizar la salida estándar *stdout* con printf(). De esta manera cuando utilicemos redirección a un fichero, las salidas del programa irán al fichero en vez de al *stdout,* pero los errores se visualizarán por la pantalla y no se incluirán en el fichero. También es preferible **perror**() a printf() por dos razones, la primera porque se utiliza el *stream* correcto (la salida estandar de errores) y la segunda es la generacion de un mensaje de error describiendo el problema (representación textual del valor actual de la variable *errno*).

errno.h contiene los códigos de error de C en Linux. Se puede obtener un listado desde consola ejecutando el comando **errno -l**