Práctica 1: Introducción a la programación de sistemas

Objetivos

- Gestión de proyectos con la utilizad `make`.
- Creación de librerías y gestión de *plugins* con `dlopen`.
- Manejo del manual del sistema.
- Uso de las funciones del API del sistema informativas y de manejo de tiempos.

Ejercicios

Ejercicio 1. API del Sistema

En cada ejercicio añadir los ficheros de cabecera (ficheros `#include`) necesarios para que no se produzcan avisos en la compilación.

1. setuid

Completad el código proporcionado en el fichero `setuid.c` para que se gestione correctamente el código de error devuelto por la llamada al sistema `setuid` siguiendo las siguientes pautas:

- En caso de error, se mostrará por la salida de error estándar el código de error generado por la llamada, tanto en su versión numérica como la cadena asociada, y el programa finalizará retornando un 1.
- En caso de éxito el programa devolverá 0.

Usar la página de manual para comprobar el propósito de `setuid`, conocer su prototipo y que ficheros de cabecera (*ficheros* `#include`) son necesarios añadir para que no se produzcan avisos en la compilación.

En relación con este código, responder a las siguientes cuestiones:

- ¿Cuál es la funcionalidad de la llamada `setuid`?
- ¿Cómo podemos comprobar, desde la linea de comandos (el *shell*), el valor devuelto tras la ejecución del programa?

2. uname

El comando del sistema `uname` muestra información sobre diversos aspectos del sistema. Escribir un programa que muestre por la salida estándar los detalles del sistema que reporta `uname` y que sea equivalente a ejecutar el comando:

```
uname -s -n -r -v -m
```

Consultar la pagina de manual de la llamada al sistema `uname()`.

- Reportar los valores obtenidos en los puestos del laboratorio.

3. sysconf

Las funciones `sysconf()` y `pathconf()` permiten consultar información sobre la configuración del sistema y el sistema de ficheros respectivamente. Escribir un programa que muestre los siguientes parámetros y reportar los valores obtenidos en los puestos del laboratorio.

- Ticks por segundo.
- El número máximo de procesos simultáneos que puede ejecutar un usuario.
- El número máximo de ficheros que puede abrir un proceso.
- El tamaño de las páginas de memoria.
- La longitud máxima de los argumentos a un programa.
- El número máximo de enlaces de un fichero en el sistema de ficheros en el que esta el directorio home del usuario.
- El tamaño máximo de una ruta en el sistema de ficheros en el que esta el directorio home del usuario.
- El tamaño máximo de un nombre de fichero en el sistema de ficheros en el que esta el directorio home del usuario.

4. uid efectivo y real

Escribir un programa `uids.c` que muestre:

- El uid efectivo y el uid real del usuario que ejecutó el programa, e indique si el correspondiente ejecutable tiene activado el bit `setuid`.
- Adicionalmente, utilizando la llamada `getpwuid()` mostrar el nombre de usuario del usuario real, su directorio *home* y el *shell* que utiliza.

5. time

La función principal para obtener la fecha actual es `time()`. En los sistemas operativos Unix y similares, la función `time()` se utiliza para obtener el tiempo actual. Más específicamente, devuelve el tiempo transcurrido en segundos desde el 1 de enero de 1970 a las 00:00:00 UTC (marca el inicio de la *epoca* Unix).

La declaración de la función `time()` en C es la siguiente:

```
#include <time.h>

time_t time(time_t *t);
```

- `time_t` es un tipo de dato que típicamente representa el tiempo en segundos.
- Si el argumento `t` es un puntero no nulo, `time()` también almacena el valor del tiempo en la ubicación apuntada por `t`.

El uso básico de la función es simplemente llamar a `time()` con un argumento `NULL`, y devolverá el tiempo actual. Por ejemplo:

```
#include <stdio.h>
#include <time.h>

int main() {
    time_t now;
    now = time(NULL);

    printf("Número de segundos desde el 1 de enero de 1970: %ld\n", now);

    return 0;
}
```

Escribir el programa `mtime.c` que:

- Obtenga la fecha actual usando `time()` y la muestre en la salida estándar usando un formato de fecha legible utilizando la función auxiliar `ctime()`.
- Calcule la fecha de hace diez días y la muestre en la salida estándar, también en formato legible usando `ctime()`.

Responder a las siguientes cuestiones:

- ¿Dónde se reserva espacio para el valor de la cadena que devuelve la función `ctime()`?
- ¿Es necesario liberar el puntero?

6. localtime

Modifica el programa `localtime.c` para que muestre la fecha y hora actuales del sistema, de acuerdo con el patrón *Hoy es Viernes, 11:40*.

7. gettimeofday

Cuando es necesario obtener la información horaria con precisión de microsegundos se puede usar `gettimeofday()`. Escribir un programa (`gettime.c`) que mida cuánto tarda un bucle de 10000 iteraciones. En cada iteración, se incrementará una variable de tipo `int` llamada `mivar` con un valor entero que se debe pasar al programa como argumento. Si no se pasa este argumento el programa mostrará un mensaje de error y terminará devolviendo el valor 1. En otro caso el programa mostrará por la salida estándar el valor final de `mivar` y el tiempo transcurrido, retornando el programa el valor 0.

Ejercicio 2. Make y creación de librerías

En el directorio `ejercicio2` del archivo `practica1.zip` contiene un programa de test en el que se utilizan dos funciones auxiliares `funaux1()` y `funaux2()` definidas en sus correspondientes ficheros fuente:

```
#include <stdio.h>
#include "milibreria.h"

int main(int argc, char** argv){
    printf("Programa de test para probar las funciones de mi libreria\n");

    int res;

    res = funaux1();
    printf("- el resultado de ejecutar la funcion 1 es %d\n",res);

    res = funaux2();
    printf("- el resultado de ejecutar la funcion 2 es %d\n",res);


    return 0;
}
```

Para compilar este programa con `gcc`, se pueden generar por separado los ficheros objetos con las funciones auxiliares, y enlazar el programa principal con ellos para obtener el ejecutable. Por ejemplo:

```
gcc -c -o fun1.o fun1src.c
gcc -c -o fun2.o fun2src.c
gcc -o test testmain.c fun1.o fun2.o
```

Si queremos distribuir estas funciones auxiliares para que sean usadas por terceros lo habitual es distribuirlas en formato de librerías.

- Construir una librería estática (`libmilibreria.a`) que incluya el código objeto de las funciones `funaux1()` y `funaux2()` y compilar nuestro programa contra esta librería en lugar de enlazarla con `fun1.o` y `fun2.o`. Crear un fichero `makefile` que automatice la creación de la librería estática y genere el ejecutable de test enlazando con dicha librería estática.
- Construir una librería dinámica (`libmilibreria.so`) que incluya el código objeto de las funciones `funaux1()` y `funaux2()` y compilar nuestro programa contra esta librería en lugar de enlazarla con `fun1.o` y `fun2.o`. Crear un fichero `makefile` que automatice la creación de la librería dinámica y genere el ejecutable de test enlazando con dicha librería dinámica.
- Fusionar en un único `makefile` la construcción de ambas librerías (estática y dinámica) y la creación de dos ejecutables de test (`appstatic` enlazado contra la librería estática y `appdynamic` contra la dinámica).

Ejercicio 3. API dlopen

La función `dlopen()` (`dlclose()`) es una función de la librería de C que permite cargar (cerrar/descargar) dinámicamente una librería compartida en tiempo de ejecución. Utilizado `dlopen()` podemos conseguir que no sea necesario cargar todas las librerías necesarias para que funcione un programa en tiempo de carga, y retrasarlo a tiempo de ejecución cuando sea necesario. Cuando `dlopen()` carga con éxito una librería devuelve un manejador que permite posteriormente referenciar a las funciones y variables de la misma. Es habitual su uso para proporcionar extensiones o plugins a un programa, ya que permite agregar funcionalidades sin necesidad de recompilarlo. Su prototipo es el siguiente:

```
void *dlopen(const char *filename, int flags);
```

El primer argumento es la ruta de la librería (hay que incluir el nombre completo), y el segundo argumento son opciones de carga. En nuestro ejemplo usaremos el flag `RTLD_LAZY`, que indica que se utilice una resolución *perezosa* (*lazy binding*) de las funciones, es decir, la resolución de las funciones se realiza solo cuando son utilizadas por primera vez. La otra opción es `RTLD_NOW`, que fuerza la resolución inmediata de todos los símbolos no definidos incluidos en la librería en el momento de carga. Esto suele aumentar el tiempo de carga inicial, pero garantiza que todos los símbolos necesarios están disponibles antes de que la librería sea utilizada, evitando errores de resolución de símbolos posteriores.

La gestión de errores en estas funciones de enlace dinámico se realiza con la función auxiliar `dlerror()`, que devuelve un puntero a una cadena de caracteres `(char *)` que describe el último error que ocurrió en llamadas al API `dlopen`. Si no ha habido errores desde la última llamada a `dlerror()`, devuelve `NULL`. El uso típico de esta función es el siguiente:

- Antes de llamar a `dlopen()`, `dlsym()`, o `dlclose()`, se suele llamar a `dlerror()` para limpiar cualquier mensaje de error anterior: `dlerror()` nos devuelve el último mensaje de error, y borra ese error, por lo que una llamada a `dlerror()` posterior devolverá `NULL` si no ha ocurrido un nuevo error.
- Después de llamar a `dlopen()`, `dlsym()`, o `dlclose()`, se debes verificar si ocurrió un error llamando a `dlerror()`. Si `dlerror()` devuelve un valor distinto de NULL, entonces ocurrió un error, y la cadena devuelta describe el problema.

Un ejemplo sencillo de uso es el siguiente:

```
void *manejador;

dlerror(); // limpiamos errores previos
manejador = dlopen("./libejemplo.so", RTLD_LAZY);
if (!manejador) {
    fprintf(stderr, "%s\n", dlerror());
    exit(EXIT_FAILURE);
}
```

Para buscar la dirección de un símbolo (una función o una variable), en una librería compartida previamente cargada con `dlopen()` se utiliza la función `dlsym(3)`. El prototipo es el siguiente:

```
void *dlsym(void *handle, const char *symbol);
```

El primer argumento es el manejador que nos devuelve `dlopen()` y el segundo argumento es el nombre del simbolo a buscar. Si el símbolo se encuentra nos devuelve un puntero al símbolo. Si no se encuentra, `dlsym()` devuelve `NULL`.

Modificar el programa de test del Ejercicio 2 para que se cargue dinámicamente la librería dinámica con `dlopen()` en lugar de enlazarlo con la librería dinámica `libmiliberia.so`. Para compilar el programa solo hay que enlazarlo con la librería `libdl` que da soporte a la carga dinámica. Si nuestro nuevo programa de test se llama `testmaindlopen.c`, podemos compilarlo desde la línea de comandos con:

```
gcc -o testdlopen testmaindlopen.c -ldl
```