Práctica 1.1: Repaso de aspectos sobre sistemas operativos y programación en C. Manejo de hilos.

Índice

1 Objetivos

El principal objetivo de esta práctica es reforzar aspectos de programación en C sobre Linux, haciendo hincapié en algunas cuestiones esenciales de la biblioteca estándar de "C" y llamadas disponibles en el sistema GNU/Linux. En particular se incide en lo siguiente:

- El uso de la función getopt() de la biblioteca estándar de "C" para procesamiento de opciones de línea de comando
- La utilización de las llamadas al sistema de Linux para acceso a ficheros
- La creación de procesos y ejecución de programas mediante las llamadas `fork()` y `exec*()`
- La creación y manejo básico de hilos con la biblioteca POSIX Threads

Asimismo esta práctica servirá para familiarizarse con los *proyectos C con Makefile*, que se utilizarán en el laboratorio para simplificar la compilación de las prácticas. El primer ejercicio de esta práctica introduce el concepto de este tipo de proyecto así como el uso básico de la herramienta GNU make.

El archivo ficheros_p1-1.tar.gz contiene una serie de proyectos C, que o bien contienen el código de ejemplo de alguno de los ejercicios, o han de emplearse como punto de partida para el desarrollo de ejercicios de esta práctica.

2 Ejercicios

Ejercicio 1: Introducción a GNU Make y Proyectos con Makefile

Es frecuente durante el desarrollo del proyecto que haya que compilar dicho proyecto multitud de veces. Además, el proyecto va cambiando, añadiéndose ficheros, bibliotecas con las que se debe enlazar, ejecutables y bibliotecas que se deben generar, etc. Repetir en cada ocasión las llamadas al compilador `gcc` para construir cada objetivo del proyecto (p.ej., biblioteca, fichero objeto o ejecutable) es ineficiente y propenso a errores. La popular herramienta Make, diseñada

para entornos UNIX, tiene como objetivo facilitar y, en cierta medida, automatizar este proceso de compilación del proyecto.

Hay distintas versiones de la herramienta Make, sin embargo todas ellas funcionan de un modo muy parecido. Cuando se invoca, Make interpreta un fichero de texto con una sintaxis especial y nombre preestablecido (p.ej. *makefile*), que le indica los pasos que debe seguir para compilar el proyecto. Una vez confeccionado el *makefile* repetir la compilación es tan sencillo como ejecutar `make` desde el terminal.

En esta asignatura emplearemos la implementación de Make de GNU, que se encuentra instalada en el laboratorio. Cuando el usuario invoca el comando `make` desde el *shell*, la herramienta busca en primer lugar un fichero que se llama `GNUmakefile`. Si no se encuentra se busca un fichero llamado `makefile`, y si por último no se encontrase, se buscaría el fichero `Makefile`. Si no se encuentra en el directorio actual ninguno de esos tres ficheros, se producirá un error y make no continuará:

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

En caso contrario, `make` procesará el archivo de texto e invocará el conjunto mínimo de reglas requeridas para construir nuestro proyecto.

Para ilustrar el flujo de trabajo de make, consideremos el código de ejemplo que se encuentra en el directorio *ejercicio1*, que constituye un proyecto en C con Makefile. Este tipo de proyectos están formados por el código fuente C del proyecto (ficheros .c y .h) más un fichero *Makefile* , que contiene las reglas necesarias para construir el objetivo (*target*) del proyecto. En este caso particular el proyecto contiene un único fichero .c (*ejemplo.c*) –con el código de un programa muy simple que acepta dos argumentos obligatorio–, y un Makefile con las reglas necesarias para generar el ejecutable del programa y limpiar los ficheros resultantes de la compilación.

Normalmente, las reglas del fichero *Makefile* garantizan que si solo se ha modificado un archivo C desde que se invocó `make` por última vez, una nueva invocación de `make` realizará la compilación solo para el archivo C modificado y la etapa final de enlazado.

Para familiarizarte con GNU Make, abre una ventana de terminal en Linux, y realiza las siguientes acciones desde el directorio *ejercicio1* (usar `cd <ruta_ejercicio_1>`), respondiendo a las preguntas que se plantean:

- Ejecuta `make` en la línea de comandos y comprueba las órdenes que ejecuta para construir el proyecto. ¿qué comando(s) se invoca(n)? ¿Se ha generado algún fichero ejecutable como resultado? Ejecuta `ls` para ver el contenido del directorio.
- Tras construir el proyecto, ejecuta los siguientes comandos, e indica la salida generada por pantalla. ¿A que se debe este comportamiento? Consulta el código fuente usando un editor de textos.
  - `./example`
  - `./example John`
  - `./example John Smith`
- Vuelve a ejecutar el comando `make` en más ocasiones. ¿Se vuelve a generar el fichero ejecutable invocando al compilador `gcc`? ¿Por qué?
  - Marca el fichero *example.c* como modificado ejecutando `touch example.c`. Después ejecuta de nuevo `make` . ¿Qué diferencia hay con la última vez que lo ejecutaste? ¿Por qué?
- Ejecuta la orden make clean. ¿Qué ha sucedido?

En este curso, la mayor parte de los proyectos C de ejemplo proporcionados con las distinas prácticas se acompañan de un fichero *Makefile* específico para el proyecto. Por lo general, para compilar el proyecto asociado bastará teclear `make` en la terminal. Cabe destacar que durante el desarrollo de la asignatura los estudiantes no tendrán que elaborar ficheros *Makefile* de cero, sino adaptar mínimamente ejemplos sencillos ya proporcionados. Por lo tanto en este documento no procederemos a introducir la sintaxis de GNU make. No obstante, esta sintaxis está descrita en el tutorial disponible en este enlace.

Ejercicio 2: Escritura y lectura de cadenas de caracteres en ficheros

En este ejercicio se han de desarrollar dos programas sencillos `write_strings.c` y `read_strings.c` que permitan respectivamente escribir y leer de un fichero un conjunto de cadenas de caracteres de longitud variable terminadas por `'\0'`. Dicho carácter terminador deberá almacenarse en el fichero con el resto de caracteres de cada cadena. Para el desarrollo de los dos programas se utilizará la función de la biblioteca estandar `malloc()`, así como las siguientes llamadas al sistema de Linux: `open()`,`close()`, `read()`, `write()`, `seek()` y `malloc()`

El programa `write-strings.c` aceptará como primer parámetro el nombre de un fichero de texto donde se escribirán los strings pasados a continuación a la línea de comandos (argumento 2, argumento 3, etc.) - véase ejemplo de ejecución más abajo. Si el fichero destino existe, el programa reescribirá su contenido.

El programa `read-strings.c` aceptará como parámetro el nombre del fichero de texto donde se almacenen las cadenas de caracteres terminadas en `'\0'`. Este programa leerá las cadenas y las imprimirá por pantalla separadas por un salto de línea, como se muestra en el siguiente ejemplo de ejecución:

```
## Write strings to file
usuarioso@debian:~/exercise2$ ./write_strings out London Paris Madrid Barcelona Berlin Lisbon

## Check whether file structure is correct (null-terminated strings)
usuarioso@debian:~/exercise2$ $ xxd out
00000000: 4c6f 6e64 6f6e 0050 6172 6973 004d 6164  London.Paris.Mad
00000010: 7269 6400 4261 7263 656c 6f6e 6100 4265  rid.Barcelona.Be
00000020: 726c 696e 004c 6973 626f 6e00            rlin.Lisbon.

## Read strings from file
usuarioso@debian:~/exercise2$ ./read_strings out
London
Paris
Madrid
Barcelona
Berlin
Lisbon
```

Por simplicidad para la implementación del programa `read-strings.c`, se ha de desarrollar una función auxiliar `char* loadstr(int fd)`. Esta función lee una cadena de caracteres terminada en `'\0'` del fichero cuyo descriptor se pasa como parámetro, reservando dinámicamente la cantidad de memoria adecuada para la cadena leída y retornando dicha cadena. La función tendrá que averiguar primero el número de caracteres de la cadena que comienza a partir de la ubicación actual del puntero de posición del fichero, leyendo carácter a carácter. Una vez detectado el caracter terminador, restaurará el indicador de posición del fichero (moviéndolo hacia atrás) y, finalmente realizará una lectura de la cadena completa.

Ejercicio 3 La función `getopt()` de la biblioteca estándar de "C"

En este ejercicio, trabajaremos el uso de `getopt()` una herramienta esencial para el procesado de opciones en línea de comando. El objetivo del ejercicio es completar el código del fichero `getopt.c` para que sea capaz de procesar las opciones `-e` y `-l` tal y como indica el uso del programa, que puede consultarse con la opción -h:

```
$ make
$ ./getopt -h
Usage: ./getopt [ options ] title

    options:
        -h: display this help message
        -e: print even numbers instead of odd  (default)
        -l length: length of the sequence to be printed
        title: name of the sequence to be printed
```

Una vez completado, el programa deberá imprimir una secuencia de `length` números (10 por defecto; podemos cambiarlo con la opción `-l`) impares (por defecto) o pares si se incluye la opción `-e`. Los argumentos `-l length` y `-e` son opcionales, pero el argumento `title` siempre debe estar presente en la línea de comando.

Ejemplos de salidas para diferentes combinaciones de entrada:

```
$ ./getopt hola
Title: hola
1 3 5 7 9 11 13 15 17 19

./getopt -l 3 hola1
Title: hola1
1 3 5

./getopt -l 4 -e hola2
Title: hola2
2 4 6 8
```

Para la realización de este ejercicio es necesario familiarizarse con la función `getopt()` consultando la página de manual de `getopt()`: `man 3 getopt`

- `int getopt(int argc, char *const argv[], const char *optstring);`

  La función suele invocarse desde `main()`, y sus dos primeros parámetros coinciden con los argumentos `argc` y `argv` pasados a `main()`. El parámetro `optstring` sirve para indicar de forma compacta a `getopt()` cuáles son las opciones que el programa acepta –cada una identificada por una letra–, y si éstas a su vez aceptan parámetros obligatorios u opcionales.

Deben tenerse en cuenta las siguientes consideraciones:

1. La función `getopt()` se usa en combinación con un bucle, que invoca tantas veces la función como opciones ha pasado el usuario en la línea de comandos. Cada vez que la función se invoca y encuentra una opción, `getopt()` retorna el caracter correspondiente a dicha opción. Por lo tanto, dentro del bucle suele emplearse la construcción *switch-case* de C para llevar a cabo la identificación de las distintas opciones. Es aconsejable no realizar el procesamiento de nuestro programa dentro del bucle, sino únicamente identificar las opciones que pasa el usuario y dar valor a variables/flags que serán utilizadas en el resto de nuestro código para decidir el comportamiento que debe tener.
2. Un aspecto particular de la función `getopt()` es que establece el valor de distintas variables globales tras invocarse, siendo las más relevantes las siguientes:
   - `char* optarg`: almacena el argumento pasado a la opción actual reconocida, si ésta acepta argumentos. Si la opción no incluye un argumento, entonces `optarg` se establece a `NULL`
   - `int optind`: representa el índice del siguiente elemento en el `argv` (elementos que quedan sin procesar). Se usa frecuentemente para procesar argumentos adicionales del programa que no están asociados a ninguna opción. Veremos un ejemplo de ello en la práctica.

Para completar el código, incluye las opciones `-l` y `-e` en la llamada a `getopt()` y completa la estructura *switch-case* para modificar los valores por defecto de la variable `options`. Para leer el valor numérico asociado a la opción `-l`, deberás utilizar la variable global `optarg`, teniendo en cuenta que esta variable es una cadena de caracteres (tipo `char *`) y, sin embargo, queremos almacenar la opción como un número entero (tipo `int`). Consulta el uso de la función `strtol()` en el manual (`man 3 strtol`) para saber cómo realizar esa conversión.

Asimismo, dado que el argumento `title` no será procesado por `getopt()` (pues no está precedido por una marca de opción al estilo `-l`), deberemos continuar el procesamiento de la cadena de entrada tras el bucle `for`. Para ello, se usará la variable `optind` junto con `argv` para almacenar el valor de la cadena de caracteres que será el título de nuestra secuencia.

Completa el código y responde a las siguientes preguntas:

1. ¿Qué cadena de caracteres debes utilizar como tercer argumento de `getopt()`?
2. ¿Qué línea de código utilizas para leer el argumento `title`?

Ejercicio 4: Creación de procesos y ejecución de programas

Desarrollar un programa `run_commands` que ejecute comandos especificados por el usuario, y espere a su terminación. Se proporciona un esqueleto de código con un *Makefile*, así como dos ficheros de entrada para comprobar el correcto funcionamiento del programa a desarrollar.

En el fichero *run_commands.c* proporcionado se incluye un código de prueba, que habrá que modificar para desarrollar la funcionalidad deseada. Este programa de partida acepta un único argumento donde se especifica un comando. El programa analiza dicho comando, y construye un array de cadenas de caracteres acabadas en `NULL` (formato de `argv`), que posteriormente se imprime por pantalla, liberando adecuadamente la memoria reservada con `malloc()`. El propósito de este programa es ilustrar el uso de la función `parse_command()` proporcionada, que deberá estudiarse y reusarse en la implementación del ejercicio.

El programa `run_commands` a desarrollar reconocerá un conjunto de opciones en la línea de comandos, que se deben procesar usando la función `getopt()`, empleada en prácticas previas. A continuación se presentan las opciones que aceptará el programa, que se recomienda implementar y probar en el orden en el que se describen:

- `-x <comando>`: Cuando el programa se invoque con esta opción, se creará un proceso hijo que ejecutará dicho comando. Para ello, el programa principal invocará la función `launch_command()` –a implementar–, que creará un proceso hijo con `fork()`, y hará que este ejecute el comando pasado como parámetro usando `execvp()`. La función devolverá el PID del proceso hijo, sin esperar a que termine su ejecución. El programa principal se encargará de esperar la terminación del proceso hijo creado. La función `launch_command()` tendrá el siguiente prototipo:

  ```
  pid_t launch_command(char** argv);
  ```

- `-s <fichero>`: Esta opción permitirá al usuario indicar como argumento la ruta de un fichero con comandos a ejecutar. Este fichero será interpretado por líneas, tomando cada línea como un comando a ejecutar con la función `launch_command()`. Los comandos se ejecutarán de forma secuencial, esperando a que un comando termine antes de ejecutar el siguiente. **Sugerencia:** usar `fgets()` para leer del fichero por líneas. Consultar `man 3 fgets`

Ejemplo de ejecución

```
# Testing -x switch
$ ./run_commands -x ls
Makefile        run_commands    run_commands.c  run_commands.o  test1           test2

$ ./run_commands -x "echo hello"
Hello

# Testing -s switch
$ ./run_commands -s test1
@@ Running command #0: echo hello
hello
@@ Command #0 terminated (pid: 1439, status: 0)

@@ Running command #1: sleep 2
@@ Command #1 terminated (pid: 1440, status: 0)
@@ Running command #2: ls -l
total 88
-rw-r--r--@ 1 usuarioso  usuarioso   267 Oct 20 11:34 Makefile
```

```
-rwxr-xr-x  1 usuarioso  usuarioso  9960 Oct 20 11:58 run_commands
-rw-r--r--  1 usuarioso  usuarioso  4332 Oct 20 11:57 run_commands.c
-rw-r--r--  1 usuarioso  usuarioso  8984 Oct 20 11:58 run_commands.o
-rw-r--r--@ 1 usuarioso  usuarioso  31 Oct 20 11:34 test1
-rw-r--r--@ 1 usuarioso  usuarioso  41 Oct 20 11:46 test2
@@ Command #2 terminated (pid: 1443, status: 0)
@@ Running command #3: false
@@ Command #3 terminated (pid: 1444, status: 256)
```

Una vez acabado el desarrollo del programa `run_commands`, responde a las siguientes preguntas:

1. Al usar la opción `-x` del programa, el comando indicado como argumento se pasa encerrado entre comillas dobles en el caso de que este, a su vez, acepte argumentos, como por ejemplo `ls -l`. ¿Qué ocurre si el argumento de `-x` no se pasa entrecomillado? ¿Funciona correctamente el lanzamiento del programa `ls -l` si se encierra entre comillas simples en lugar de dobles? **Nota:** Para ver las diferencias prueba a ejecutar el siguiente comando: `echo $HOME`
2. ¿Es posible utilizar `execlp()` en lugar de `execvp()` para ejecutar el comando pasado como parámetro a la función `launch_command()` ? En caso afirmativo, indica las posibles limitaciones derivadas del uso de `execlp()` en este contexto.
3. ¿Qué ocurre al ejecutar el comando `"echo hello > a.txt"` con `./run_commands -x` ? ¿y con el comando `"cat run_commands.c | grep int"` ? En caso de que los comandos no se ejecuten correctamente indica el motivo.

Ejercicio 5: Creación y paso de parámetros a hilos.

En este ejercicio vamos a usar la biblioteca de pthreads, por lo que será necesario compilar y enlazar con la opción `-pthread`.

Escribir un programa *hilos.c* que va a crear hilos cuya funcionalidad vendrá determinada por los argumentos que se le pasen en la creación. Los hilos recibirán como argumentos el puntero a una estructura que contenga dos campos: un entero, que será el número de hilo, y un caracter, que indicará si el hilo es prioritario (P) o no (N).

El programa deberá crear una variable para el argumento de cada hilo usando memoria dinámica, inicializar dicha variable con el número de hilo y su prioridad (los pares serán prioritarios y los impares no lo serán), crear los hilos y esperar a que finalicen.

Cada hilo copiará sus argumentos en variables locales, liberará la memoria dinámica reservada para los mismos, averiguará cuál es su identificador e imprimirá un mensaje con su identificador, el número de hilo y su prioridad.

El alumno debe consultar las páginas de manual de: `pthread_create`, `pthread_join`, `pthread_self`.

Probar a crear solamente una variable para el argumento de todos los hilos, dándole el valor correspondiente a cada hilo antes de la llamada a `pthread_create`. Explicar qué sucede y cuál es la razón.