SISTEMAS OPERATIVOS

Sesión 4: Compilación y depuración

Lectura previa: este documento debe ser leído y entendido antes de empezar la sesión de laboratorio.
Tiempo estimado: entre 1 hora y 1h30'.

Introducción

Linux incorpora herramientas para la creación de programas y su posterior depuración. En concreto, incorpora un entorno para facilitar el mantenimiento de aplicaciones creando pequeños scripts, llamados Makefiles, utilizados para compilar programas de una forma eficiente y sencilla.
Por otro lado, también se proporcionan herramientas de depuración necesarias en la creación de programas. Estas herramientas están pensadas para ayudar al programador en la eliminación de fallos de programación y funcionamiento de programas.

Objetivos de la práctica

En esta práctica se pretende que el alumno adquiera los conocimientos necesarios para facilitar el trabajo en las siguientes prácticas. En concreto, el alumno deberá saber:

- Compilar i montar programas en C
- Concepto de tratamiento de errores
- Crear y modificar Makefiles
- Depurar programas mediante Strace

Desarrollo de programas C

Un programa en C está formado por un conjunto de funciones. En todo programa hay una función principal, la función **main()**, que le indica al compilador cuál es la primera sentencia a ejecutar. A veces, por cuestiones de claridad, el código que formará un mismo ejecutable está separado entre varios ficheros fuente, como en el ejemplo de la figura 1. En el programa tenemos un código sencillo compuesto por un único fichero, ejemplo.c. Este fichero tiene una función main() y dos funciones auxiliares suma() e imprime_num(). Las tres funciones están en el mismo fichero fuente. En la figura 2 tenemos el mismo ejemplo repartido en tres ficheros fuentes. En el programa principal, ejemplo1.c ahora sólo tenemos la función main().
Las otras dos funciones (suma() e imprime_num()) las hemos puesto en ficheros separados.
- NOTA: Tened en cuenta que para poder utilizar las funciones suma() e imprime_num() desde el programa ejemplo1.c, es necesario haber especificado los prototipos de las funciones.

A partir de ahora trabajaremos con el programa ejemplo1.c para explicar el desarrollo básico de un programa en C

Figura 1.- Contenido de ejemplo1.c:

```
void suma(int x,int y, int *z)
{
       *z=x+y;
}
void imprime_num(int x)
{
       char buffer[20];
       sprintf(buffer, “%d\n”, x);
       write(1, buffer, strlen(buffer));
}
int main(int argc,char *argv[])
{
       int a,b,c;
       a=atoi(argv[1]);
       b=atoi(argv[2]);
       suma(a,b,&c);
       imprime_num(c);
}
```

Figura 2.-

Contenido de f1.c

```
void suma(int x,int y, int *z)
{
       *z=x+y;
}
```

Contenido de f2.c

```
void imprime_num(int x)
{
       char buffer[20];
       sprintf(buffer, “%d\n”, x);
       write(1, buffer, strlen(buffer));
}
```

Contenido de ejemplo1.c

```
void suma(int x,int y, int *z); /* definición de los prototipos */
void imprime_num(int x);         /* de las funciones suma e imprime_num */
int main(int argc,char *argv[])
{
       int a,b,c;
       a=atoi(argv[1]);
       b=atoi(argv[2]);
       suma(a,b,&c);
       imprime_num(c);
}
```

Ficheros de cabecera

En C podemos especificar los que llamaremos “ficheros de cabeceras” o *include files*. Estos ficheros se componen de prototipos de funciones y definición de tipos de datos que queremos utilizar en otros ficheros, de forma que podemos reutilizarlos.
Los ficheros de cabeceras son ficheros con extensión .h (de *header*) y se “incluyen” en tiempo de preprocesado en el programa en el cual necesitaríamos especificar las funciones o tipos de datos: ésto se le indica mediante la directiva *#include*. Si el fichero .h no está en los directorios del sistema /usr/include, /usr/local/include, etc, deberemos especificar su ubicación y poner el nombre entre comillas. Si está en algún directorio estándar lo pondremos de la siguiente forma: *#include <nombre_fichero.h>*

El ejemplo anterior quedaría de la siguiente forma:

Contenido de f1.c
```
#include “auxiliares.h”
void suma(int x,int y, int *z)
{
	*z=x+y;
}
```

Contenido de f2.c
```
#include “auxiliares.h”
#include <stdio.h> /* debido a la función sprintf */
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h> /* debido a la function strlen */
	/* para saber en que fichero de cabecera está una llamada a sistema */
	/* o instrucción se consulta el man, sección 2 (llam.) o 3 (instr.) */
	/* por ejemplo man 2 write */
void imprime_num(int x)
{
	char buffer[20];
	sprintf(buffer, “%d\n”, x);
	write(1, buffer, strlen(buffer));
}
```

Contenido de ejemplo1.c
```
#include “auxiliares.h”
int main(int argc,char *argv[])
{
	int a,b,c;
	a=atoi(argv[1]);
	b=atoi(argv[2]);
	suma(a,b,&c);
	imprime_num(c);
}
```

Contenido de auxiliares.h
```
void suma(int x,int y, int *z);
void imprime_num(int x);
```

Compilación

Para generar un ejecutable a partir de 1 o N ficheros fuente es necesario realizar dos pasos: compilación y montaje. La compilación es la traducción de un fichero fuente a un código intermedio. Este proceso necesita el fichero fuente (.c) y genera un fichero objeto (.o).

El comando ***cc*** (C compiler) permite compilar y montar los programas desarrollados en C. Si queremos cambiar el nombre del ejecutable o bien solamente compilar el programa, necesitaremos recurrir a las siguiente opciones:

- **-c:** Suprime la fase de montaje, no borra ningún fichero objeto producido (sin esta opción los ficheros objeto se borran).
- **-o *nombre*:** Usa el ***nombre*** indicado, en vez de a.out, para el fichero ejecutable que se produzca. Resulta recomendable usar esta opción pues un ejecutable *a.out* de otro programa desaparecerá al compilar de nuevo.
- **-S:** Al compilar el fichero en lenguaje C se genera un fichero en lenguaje ensamblador como paso intermedio. Normalmente este fichero se borra. La opción -S evita que estos ficheros se borren. Los ficheros ensamblador tienen la extensión “.s”.

- **-D__STDC__ :**El compilador efectúa una comprobación de tipos para fuciones definidas según las normas ANSI.

En el ejemplo que estamos utilizando deberíamos hacer:

```
> cc -c ejemplo1.c
> cc -c f1.c
> cc -c f2.c
```

Si el ***cc*** encuentra errores nos los indicará diciendo la línea y el tipo de error. Después de compilar los tres programas tendríamos los ficheros ejemplo1.o, f1.o, y f2.o.
Podemos utilizar cc -v para ver las opciones que utiliza cc para cada fase de la compilación.

Montaje

Para generar el ejecutable necesitamos realizar la fase de montaje (en inglés, *linkage*). Utilizaremos también el *cc*. Necesitamos indicarle todos los ficheros objeto que forman el ejecutable, en este caso ejemplo1.o, f1.o, y f2.o.
El ***cc*** genera un ejecutable con un nombre por defecto (siempre el mismo), si el usuario no indica otra cosa. El nombre es *a.out*.

```
> cc ejemplo1.o f1.o f2.o
```

Se puede utilizar la opción -o para indicar el nombre del ejecutable.

```
> cc -o ejemplo1 ejemplo1.o f1.o f2.o
```

Además de poder hacerse directamente, en el mismo proceso de compilación, con la utilidad ***cc***, puede utilizarse el comando *ld* (linker o montador).

Ficheros de cabecera

Para facilitar el manejo de constantes y declaraciones de tipos, entre otras cosas, el lenguaje C ofrece la directiva #include “nombre.h”. Cualquier línea que comience por #include será sustituida por el contenido del fichero indicado por “nombre.h”. De este modo, las librerías suelen tener asociados ficheros en los que se especifican las interficies de las rutinas, constantes y tipos de datos que ofrece la librería. Los programas en C que las usen deben añadir las líneas que incluyan estos ficheros.
Si el nombre del fichero está delimitado por <> (por ejemplo #include <fcntl.h>), se está haciendo referencia a un fichero de cabecera del sistema. Dichos ficheros se encuentran en /usr/include.

Ejemplo

Vamos a desarrollar un programa que imprima los parámetros que se le pasan en la línea de comandos.

```
> cat prog.c
main(int argc, char *argv[]){
       int i;
       for (i=0; i<argc; i++)
       printf(“%d %s \n”,i,argv[i]);
}
> cc prog.c -o prog
> prog p1 p2 p3
0 prog
1 p1
2 p2
3 p3
>
```

```
> cat cabecer.h
#define TRUE 1
#define FALSE 0
```

La función main( ) tiene dos parámetros denominados *argc* y *argv*. El parámetro *argv* es un vector de strings en el cual se indica el nombre con el que se invocó el programa y los parámetros de la línea de comandos. La cantidad de elementos que tiene *argv* está contenida en *argc*.
Podríamos pensar en desarrollar una función que dado un elemento de argv compruebe si es una opción (comenzará por un guión) .

A continuación podemos compilar, montar y ejecutar:

```
> cat prog2.c
#include “cabecer.h”
main(int argc, char *argv[]){
       int i;
       printf(“programa: %s \n”,argv[0]);
       for (i=1; i<argc; i++){
       if (es_opcion(argv[i])==TRUE)
              printf(“opcion: %s \n”,&argv[1][1]);
       else
              printf(“argumento: %s \n”,argv[i]);
       }
}
> cat param.c
#include “cabecer.h”
int es_opcion(char *param){
       if (param[0]==’-’) return(TRUE);
       else return(FALSE);
}
> cc -c prog2.c
> cc -c param.c
> cc -o prog2 prog2.c param.c
> prog2 p1 -opc p2
programa: prog2
argumento: p1
opcion: opc
argumento: p2
prompt%
```

Ayudas a la compilación y montaje: Makefiles

La tarea de compilar y montar los programas puede resultar tediosa, sobre todo en el caso de tener un gran número de ficheros. La utilidad make permite compilar y montar de manera automática. Además de permitirnos especificar todos los pasos para generar uno o más ejecutables de forma sencilla, se encarga de gestionar qué ficheros han sido modificados y necesitan ser compilados de nuevo.
La utilidad *make* utiliza un fichero en el que se le indica los objetivos y cuáles son los pasos o reglas a seguir para conseguirlo. La utilidad make también puede recibir opciones.
El nombre del fichero que contiene las reglas se indicada con la opción -f nombreFichero. Por defecto, el nombre del fichero es makefile o Makefile (se elige el primero en caso de existir los dos nombres en el directorio)
Un fichero de reglas tiene el siguiente formato:
objetivo: [requisitos]
<TAB> comando
[<TAB> comando]
Tanto el objetivo como los requisitos suelen ser ficheros situados en el mismo directorio que el *Makefile*. El trabajo básico de *make* es mantener al día el objetivo,

asegurándose que los ficheros de los cuales depende existen y están también al día si aparecen, a su vez, como objetivos en alguna otra parte del *Makefile*.
Queremos recalcar que *las líneas de comandos deben comenzar por tabulador*, de este modo make puede distinguirlas de las líneas de requisitos. En caso de que no comiencen con tabulador, el programa *make* presenta el siguiente mensaje de error: *Must be a separator on line N*, que indica que en una línea cercana a la número N (no necesariamente ésta) debería comenzar con tabulador.
Make opera basándose en tres fuentes de información:

- el fichero de descripción de objetivos como este anterior
- los nombres de los ficheros implicados, sus extensiones (letras después del ".") y sus fechas de última modificación
- sus reglas internas, que aplica cuando no se especifica el comando a realizar para conseguir el objetivo.

Podemos realizar una comprobación práctica de sus reglas internas: en un directorio en el que tengamos un programa *pro.c* , si invocamos el comando *make pro* ,veremos cómo reconoce que es un programa escrito en C, y lo compila y monta correctamente, sin ni siquiera necesidad de *Makefile*.
En el ejemplo que mostramos a continuación, veremos que para conseguir el objetivo ejemplo1 es necesario tener previamente los ficheros ejemplo1.o, f1.o y f2.o. A su vez, para obtener ejemplo1.o es necesario tener ejemplo1.c y *auxiliares.h*. Una regla similar rige para la obtención de f1.o y f2.o.
[NOTA: para seguir el ejemplo debes asegurarte que no existen ya los ficheros ejemplo1.o, f1.o, f2.o y ejemplo1).

```
> cat Makefile
# Makefile de ejemplo
# las lineas que comienzan por # son comentarios
# por defecto, el primer objetivo es el que se busca conseguir
# regla (1)
ejemplo1: ejemplo1.o f1.o f2.o
	cc -o ejemplo1 ejemplo1.o f1.o f2.o
# regla (2)
ejemplo1.o: ejemplo1.c auxiliares.h
	cc -c ejemplo1.c
# regla (3)
f1.o: f1.c auxiliares.h
	cc -c f1.c
# regla (4)
f2.o: f2.c auxiliares.h
	cc -c f2.c
clean:
	rm -f ejemplo1.o f1.o f2.o ejemplo1
```

Cuando el usuario teclea make ejemplo1 el programa make busca el fichero Makefile. Lee este fichero y en él se le indica que para obtener el objetivo ejemplo1 sus requisitos (ejemplo1.o, f1.o y f2.o) deben tener fechas anteriores al fichero objetivo. El programa sigue adelante mirando si ejemplo1.o, f1.o y f2.o son, a su vez, objetivo de alguna regla. Como lo son vuelve a aplicar el paso anterior a los requisitos de cada uno de los nuevos objetivos no sin antes guardar por donde ha pasado ya. Este proceso recursivo para cuando se llega a ficheros requisito que no son objetivo de ninguna regla.
Llegado a este punto se comprueban las fechas relativas entre objetivo y requisitos. Si el objetivo es anterior a los requisitos o bien no existe, se aplica la acción o acciones que hay escritas a continuación. Si el objetivo es posterior a los requisitos se vuelve a la regla que condujo a la actual.

La segunda vez que se invocara a la utilidad make se obtiene el resultado 'ejemplo1' is up to date. Es decir, las fechas de todos los ficheros involucrados en la compilación y montaje están ordenadas correctamente.
Podemos cambiar la fecha de última modificación de un fichero mediante el comando touch (man touch) y volver a ejecutar make; se consideraría modificado y se repetiría la regla correspondiente (prueba hacerlo con alguno de ellos). Si modificáramos auxiliares.h se compilarían los ficheros objeto al detectar la diferencia de fecha.
En caso de llamar a *make* sin indicarle ningún objetivo, éste intentará resolver el primer objetivo que se encuentre en el fichero Makefile, por eso es importante ordenar las reglas de modo conveniente.
Por último, en los Makefiles se suele incluir una regla para borrar todos los ficheros temporales que se han creado al compilar y linkar (ficheros con extensión .o). Esta regla se suele llamar *clean*. Al invocar la regla clean (make clean) del ejemplo, se borrarán los ficheros: ejemplo1.o, f1.o, f2.o y ejemplo1. Es recomendable añadir esta regla siempre a los Makefiles.

Depuración de programas

STRACE

Strace es un comando que existe en las instalaciones estándar de Linux. Este comando es muy útil puesto que muestra las llamadas al sistema que se ejecutan de un determinado programa.
Para la explicación de strace, utilizaremos el siguiente código de ejemplo:

Prueba.c
```
int main ()
{
  char c;

  while (read (0, &c, 1)>0)
  {
    write (1, &c, 1);
  }
  exit (0);
}
```

Con el siguiente fichero **p** (que contiene “abc”) que redireccionaremos a la entrada estándar:

Después de compilar y linkar el programa, lo probamos:
```
> cc –o prueba prueba.c
> prueba < p
abc
```

Vemos que es un programa del estilo de cat, es decir, muestra por su salida estándar todo lo que lee por su entrada estándar.

La ejecución con strace siempre es:
```
> strace [OPCIONES]  programa [OPCIONES]
```

En el caso del programa anterior, haremos:
```
> strace prueba < p
```

La salida de strace, al principio puede ser un poco confusa. Esto es debido a que el programa hace llamadas al sistema que nosotros no hemos programado. Básicamente, lo que hace al principio es cargar las librerías de sistema necesarias para la ejecución de la aplicación. Por tanto, es importante saber donde empieza la ejecución de nuestro código. En nuestro caso, después de eliminar las llamadas al sistema que no forman parte de nuestro código, veríamos:

```
...
read(0, "a", 1)                         = 1
write(1, "a", 1a)                        = 1
read(0, "b", 1)                         = 1
write(1, "b", 1b)                        = 1
read(0, "c", 1)                         = 1
write(1, "c", 1c)                        = 1
read(0, "\n", 1)                        = 1
write(1, "\n", 1)                        = 1
read(0, "\n", 1)                        = 1
write(1, "\n", 1)                        = 1
read(0, "", 1)                          = 0
_exit(0)                                = ?
```

Cada línea nos muestra una llamada al sistema. El formato de cada línea es el siguiente:

```
LLAMADA(PARÁMETROS) =CÓDIGO_RETORNO
```

En el ejemplo anterior, la primera llamada al sistema es read. Sus parámetros son el número de canal (0), el búfer donde almacena lo que ha leído (que en este caso ya contiene el primer carácter de la entrada estándar), y el número de caracteres que tiene que leer. En este caso, el código de retorno es 1 puesto que ha leído un única carácter.

Strace tiene parámetros que modifican su comportamiento. Los más útiles son:

- -c: Cuenta el tiempo, llamadas y errores por llamada al sistema y muestra un informe al finalizar la ejecución
- -f: tracea procesos hijos a medida que se crean por procesos que se están traceando como resultado de la llamada al sistema fork.
- -e trace=lista: tracea solamente las llamadas al sistema específicas.
- -o fichero: escribe el resultado en un fichero.
- -p pid: tracea un proceso con el pid especificado.

Así, si queremos mostrar solamente el resultado de las llamadas al sistema read, tendremos que hacer:

```
> strace –e trace=read prueba < p
read(0, "a", 1)                          = 1
aread(0, "b", 1)                         = 1
bread(0, "c", 1)                         = 1
cread(0, "\n", 1)                        = 1
read(0, "\n", 1)                         = 1
read(0, "", 1)                           = 0
```

En este caso, la “a”, la “b” y la “c” que salen al principio de cada línea es el resultado de la ejecución del programa. Strace muestra su salida por la salida estándar de error, por lo tanto, el resultado del programa y del strace se mezclan.

Tratamiento de errores en UNIX

La mayoría de las llamadas a sistema en UNIX suelen retornar un valor -1 cuando se produce un error. Sin embargo existe una manera de saber que tipo de error se ha producido: por medio de la variable global **int errno;**

En la variable errno, el SO pone un número que identifica el tipo de error producido. La variable no debe declararse explícitamente, pero necesitas hacer un #include <errno.h> para poder trabajar con ella.

También se declara una serie de constantes que se pueden utilizar dentro del programa ("man errno" da la lista completa, así como "man *llamada*" da la lista de posibles errores que puede dar esa llamada: haced "man 2 write" i mirad la sección errors).
Además, el sistema ofrece la llamada sterror, que dado un número de error, devuelve un string con una frase explicativa (en inglés). Haced man strerror para más información.