Actividad 5 (v. 230318) Pablo González Nalda

Depto. de *Lenguajes y Sistemas Informáticos* lsi.vc.ehu.es/pablogn

Universidad del País Vasco Euskal Herriko Unibertsitatea

GASTEIZKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE VITORIA-GASTEIZ

El objetivo de esta actividad es estudiar la **creación de librerías estáticas y dinámicas**. Para ello vamos a compilar una librería y después añadirla a un programa principal de tres formas diferentes, observando el resultado. Como consecuencia de esta práctica, obtenemos este documento que detalla el proceso y los objetivos de aprendizaje.

El ejemplo que se va a tratar es similar a los siguientes:

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

https://renenyffenegger.ch/notes/development/languages/C-C-plus-plus/GCC/create-libraries/index

5.1. Fases de la compilación y enlazado

Antes de poder observar cómo se unen diferentes módulos, subprogramas y librerías de dos tipos, repasamos y observamos cómo se construye un programa simple del tipo *Hola, mundo*. En este caso, el compilador sustituye el `printf` de una cadena constante, es decir, que no imprime variables, por la función `puts` que escribe (`put`) una cadena en la salida estándar.

Lo primero es comprobar que podemos usar GCC escribiendo `gcc -v` (en minúsculas). Si no tenemos el compilador instalado, lo obtendremos con `apt install gcc build-essential`

Compilación (incluida fase de preprocesador) hasta obtener el ensamblador:

```
gcc -v -S -masm=intel -fverbose-asm hola.c
```

Fase de ensamblador:

```
as -v --64 hola.s -o hola.o
```

Fase de enlazado

```
gcc -o hola hola.o -v
```

Análisis del resultado

```
file hola
file hola.o
nm hola
```

Es interesante el resultado de las librerías dinámicas de este sencillo programa:

```
ldd hola
linux-vdso.so.1 (0x00007ffe6a849000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f777542b000)
/lib64/ld-linux-x86-64.so.2 (0x00007f777566e000)
```

Nos indica que tiene tres librerías de enlace dinámico. La primera es un mecanismo para acelerar ciertas llamadas al sistema, la segunda es LibC, la librería estándar para que funcione cualquier programa en C. La última, `ld-linux.so` es el cargador del programa y de

librerías dinámicas, y se encarga de preparar el programa para su ejecución. El kernel carga esta librería cargadora para ejecutar el programa.

5.2. Análisis de los programas dados

Vamos a usar un programa de ejemplo, que divide una operación en dos partes, en el programa principal y en una función que se define en otro fichero diferente al principal.

Analizamos el programa principal, que calcula π como el resultado de una serie matemática (la suma de los términos de una sucesión matemática) multiplicado por 4.

```
/////////////////////////////
/// fichero  test_pi.c para calcular el valor de Pi.
///         Se basa en la formula Pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - ... )
/// Uso:  test_pi   n
///        n es el numero de iteraciones del algoritmo para obtener la aprox. de Pi
/// retorno: Si el numero de argumentos no es correcto se retorna 1. Si no, se retorna
    0
/////////////////////////////////////////////////////////////

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "calcula_pi.h"

int main(int argc, char *argv[]) {
    double t, k = 3.0, l = -1.0;
    long i, s = 1E6;

    if (argc < 2) {
        fprintf(stderr, "%s <millones de iteraciones>\n", argv[0]);
        exit(1);
    }
    s *= atoi(argv[1]);
    printf("Iteraciomes: %s -> %12ld\n",argv[1], s);
    t = 1.0;
    for(i = 0; i < s; i++) {
        funcion(&t, &k, &l); // librerÃa
    }
    t *= 4;

    printf("Mi valor de Pi: %1.16f,  Valor de Pi en math.h: %.16f\n", t, M_PI);
    printf("Diferencia Absoluta: %.16f\n", fabs(M_PI - t));
    return 0;
}
```

Fichero 5.1: `test_pi.c`

El programa principal usa un procedimiento llamado `funcion` (en el código no se pueden poner ni tildes ni eñes) que acumula los términos de la sucesión y que declara en una cabecera `calcula_pi.h` (que no tiene el código para realizar la operación).

Por tanto, el programa principal tiene conocimiento a través de `calcula_pi.h` de que debe usar una función con esa forma, pero no su contenido o código. Lo que hace el *preprocesador* es incluir todo este fichero en la posición del `#include` de la línea 13 del programa principal.

```
1 /////////////////
  // calcula_pi.h
  //
4 // fichero de include de la biblioteca libpi
  //
  ////////////////////////////////////////////
7
  void funcion (double *t, double *k, double *l);
```

Fichero 5.2: `calcula_pi.h`

En el otro fichero `.c` se suministra la definición de la función, que consiste en calcular un nuevo término modificando las variables, pasadas por referencia a través de punteros:

```
  ////////////////////////////
  /// fichero  calcula_pi.c
3 ///
  /// Fichero para crear la biblioteca coun una funcion de prueba
  ///
6 ////////////////////////////////////////////////////////////

  #include <stdio.h>
9 #include <math.h>
  #include <stdlib.h>

12 /// Funcion de prueba de ayuda al calculo de Pi
  // se basa en la formula Pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - ... )

15 void funcion (double *t, double *k, double *l) {
               (*t) += (*l)/(*k);
               (*k) += 2.0;
18             (*l) *= -1.0;
  }
```

Fichero 5.3: `calcula_pi.c`

5.3. Inclusión de la librería como código fuente

Para incluir la librería como código fuente debemos cambiar el `include` del programa principal de la línea 13.

```
1 #include "calcula_pi.c"
```

Sólo será necesario pasar al compilador el nombre del programa principal.

```
gcc -o ppal test_pi.c
2 ./ppal 300
```

5.4. Compilación de la librería como objeto y enlazado con el principal

Se compila la librería con la instrucción `gcc -c calcula_pi.c -o calculapi.o` en la que se indica con `-o` el nombre del fichero de salida.

Comprobaremos que `calculapi.o` proporciona el código de `funcion` si listamos las referencias con `nm calculapi.o`

Podríamos obtener mucha más información activando la depuración (*debugging*) con `-g3` y desactivando la optimización del código con `-O0` (letra 'o' seguida de un cero). De esta forma la compilación sería `gcc -c -O0 -g3 calcula_pi.c -o calculapi.o`

Para construir el ejecutable por pasos, tendríamos esta lista. La opción `-fPIC` es necesaria posteriormente para crear la librería dinámica.

```
gcc -c calcula_pi.c -o calculapi.o -fPIC # resultado con nombre definido
gcc -c test_pi.c # nombre a partir del cambio de extensión
gcc -o ppal test_pi.o calculapi.o
./ppal 3000
```

En este punto podemos analizar los binarios `.o` con la instrucción `nm`:

```
809  nm test_pi.o
810  nm calculapi.o
811  nm ppal
```

Para crear el ejecutable de un tirón:

```
gcc -o p test_pi.c calcula_pi.c
```

Como vemos en las siguientes líneas, al no poner el fuente de la función nos da un error al enlazar (`collect2`). Si además pusiéramos la opción `-v` proporcionaría mucha información.

```
gcc -o p test_pi.c
/tmp/cc3upJKo.o: En la función `main':
test_pi.c:(.text+0xa7): referencia a `funcion' sin definir
collect2: error: ld returned 1 exit status
```

Todas las instrucciones que hemos usado en este punto:

```
804  gcc -v
805  gcc -c calcula_pi.c -o calculapi.o
806  nm calculapi.o
807  gcc -c -O0 -g3 calcula_pi.c -o calculapi.o
808  gcc -c test_pi.c
809  gcc -o ppal test_pi.o calculapi.o
809  nm test_pi.o
810  nm calculapi.o
811  nm ppal
812  ./ppal 3000
813  gcc -o p test_pi.c calcula_pi.c
814  gcc -o p test_pi.c
815  history
```

Hasta ahora hemos compilado la librería en un fichero e integrado su código binario en el ejecutable mediante el proceso de enlazado. En los siguientes puntos, haremos librerías que se pueden mantener en el sistema accesibles para que los ejecutables se construyan usándolas y compartiéndolas entre diferentes programas.

5.5. Creación de las librerías dinámica y estática, y enlazado

A partir del fichero `calculapi.o`, con el binario de la función `funcion` vamos a crear las dos librerías y los dos ejecutables. Un ejecutable es el que usa la estática y el otro la dinámica.

Como ejemplo de librerías estáticas y dinámicas podemos ver varias con este comando:

```
ls -l /usr/lib/libmono*
```

Empezamos con la librería estática:

```
# Crea la librería estática
ar rcs libpi.a calculapi.o

# Enlaza la estática (libpi.a -> -lpi) con el main en el ejecutable (libestaticapi)
gcc -static test_pi.c -L. -lpi -o libestaticapi
```

En este caso sólo ha incluido en la librería un objeto. El fichero `.a` podría copiarse en el directorio `/usr/lib` para hacerse accesible a todos las cuentas usuarias del sistema, por si quieren enlazar ese código o librería con su programa.

```
# Crea la librería dinámica
gcc -shared -Wl,-soname,libpi.so.1 -o libpi.so.1.0.1 calculapi.o
```

Para enlazar la librería dinámica (`libpi.so ->-lpi`) con el main en el ejecutable (libdinamicapi) hay que realizar la siguiente instrucción. El problema es que si ponemos directamente:

```
gcc test_pi.c -L. -lpi -o libdinamicapi
  /usr/bin/ld: no se puede encontrar -lpi
  collect2: error: ld returned 1 exit status
```

Necesitamos simplificar el nombre de la librería con un enlace simbólico para que la encuentre y para que si cambia la librería no cambiemos su nombre sino el enlace simbólico.

```
ln -s libpi.so.1.0.1 libpi.so
gcc test_pi.c -L. -lpi -o libdinamicapi
```

Ahora el ejecutable está enlazado correctamente con la librería dinámica, pero no la encuentra cuando preguntamos por sus librerías.

```
ldd libdinamicapi
  linux-vdso.so.1 (0x00007ffc97f67000)
  libpi.so.1 => not found
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff1d58f2000)
  /lib64/ld-linux-x86-64.so.2 (0x00007ff1d5b30000)
```

Lo solucionamos haciendo otro enlace simbólico que tenga como nombre el `soname` de la librería, su nombre integrado.

```
ln -s libpi.so.1.0.1 libpi.so.1
```

De esta forma ya se puede cargar la librería desde el programa.

```
ldd libdinamicapi
  linux-vdso.so.1 (0x00007ffe6a849000)
  libpi.so.1 => ./libpi.so.1 (0x00007f7775662000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f777542b000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f777566e000)
```

La librería dinámica tiene estar definida en la variable PATH de las librerías dinámicas para que se pueda arrancar el ejecutable que la usa. En caso contrario da error:

```
echo $LD_LIBRARY_PATH
./libdinamicapi 3000
  ./libdinamicapi: error while loading shared libraries: libpi.so.1: cannot open shared
      object file: No such file or directory
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
  ./libdinamicapi 3000
  Mi valor de Pi: 3.1419258758397897,  Valor de Pi en math.h: 3.1415926535897931
  Diferencia Absoluta: 0.0003332222499965
```

Reunimos todas las instrucciones para crear la librería dinámica:

```
$ echo $LD_LIBRARY_PATH
/opt/ros/noetic/lib
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
$ echo $LD_LIBRARY_PATH
.:/opt/ros/noetic/lib
$ gcc -c calcula_pi.c -o calculapi.o -fPIC
$ gcc -shared -Wl,-soname,libpi.so.1 -o libpi.so.1.0.1 calculapi.o
$ gcc test_pi.c -L. -lpi -o libdinamicapi
/usr/bin/ld: no se puede encontrar -lpi
collect2: error: ld returned 1 exit status
$ ldd libdinamicapi
ldd: ./libdinamicapi: No existe el archivo o el directorio
$ ln -s libpi.so.1.0.1 libpi.so.1
$ ln -s libpi.so.1 libpi.so
$ gcc test_pi.c -L. -lpi -o libdinamicapi
$ ldd libdinamicapi
linux-vdso.so.1 (0x00007ffefc7d1000)
libpi.so.1 => ./libpi.so.1 (0x00007f25c885d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f25c861b000)
/lib64/ld-linux-x86-64.so.2 (0x00007f25c8869000)
```

5.6. Análisis de los resultados de los procedimientos anteriores

```
# Medida del tiempo de ejecución
time ./libdinamicapi 3000
time ./libestaticapi 3000
time ./ppal 3000
```

Podemos aplicar `strace` y `ltrace` vistas en la Actividad 4 a los programas.

Cuando ya hemos creado los ejecutables con librería dinámica, con estática y sin librerías, comprobamos que el uso de librerías dinámicas ralentiza ligeramente la ejecución.

Si hacemos `ls -l` comprobamos que el ejecutable con librería estática es mucho más grande que la librería dinámica y el ejecutable que la usa. El ejecutable creado sin librerías incluye todo en menos de 16KB.

Evidentemente, la ventaja la obtendremos en proyectos más grandes al estructurar la generación de diferentes partes del programa que compilamos con gestores como `make` (ver http://www.thegeekstuff.com/2010/08/make-utility/), que es la herramienta más importante de *automatización de la construcción del software*. Para aprender más: https://en.wikipedia.org/wiki/List_of_build_automation_software

Existen muchas herramientas para examinar el contenido de ejecutables y librerías. Las más importantes: `file`, `nm`, `objdump`, `ldd` y `readelf`. Es muy interesante **aplicar las anteriores instrucciones**, principalmente `ldd` por ser la más sencilla y que nos indica si el ejecutable usa librerías dinámicas. En la salida de `nm` con el ejecutable estático podemos ver que incluye todas las funciones de `libc` para no incluirlas dinámicamente.

```
file libestaticapi
file libdinamicapi
file ppal
nm libestaticapi # salida larga por incluir funciones de libc
nm libdinamicapi
nm ppal
ldd libestaticapi
ldd libdinamicapi
ldd ppal
```

Por moderar la extensión del documento no incluimos la salida de estos comandos con todos los ficheros obtenidos (objeto, ejecutables y librerías), en los que se observa la estructura de elementos usados en la fase de enlazado. Más información en http://www.thegeekstuff.com/2017/01/gnu-binutils-commands/ y `apt search binutils` para diferentes arquitecturas, y `apt show binutils` para una descripción.

Librerías dinámicas de nuestro programa:

```
$ ldd libdinamicapi
linux-vdso.so.1 (0x00007ffefc7d1000)
libpi.so.1 => ./libpi.so.1 (0x00007f25c885d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f25c861b000)
/lib64/ld-linux-x86-64.so.2 (0x00007f25c8869000)
```

`ld-linux` es el programa *cargador* que ejecuta el código del ejecutable y carga las librerías dinámicas usando la llamada al sistema `mmap`, que *mapea* o coloca el fichero en la memoria del proceso:

```
$ strace ./libdinamicapi
```

5.7. Reubicación

En http://www.thegeekstuff.com/2011/10/gcc-linking/ encontramos un ejemplo parecido, en el que no se incluye la declaración de `funcion` con una directiva `#include` sino con la palabra reservada `extern`. Lo que añade ese artículo es un análisis del método de *reubicación*. Como su adaptación a este ejemplo es complicada, recomendamos leer la sección CODE RELOCATION de dicha página y comprobar que se obtienen dichas salidas.

5.8. Conclusiones

En esta práctica hemos aprendido las tres formas de crear un programa con librerías, incluido en el ejecutable, con librerías estáticas y con librerías dinámicas. Hemos observado las diferencias en tamaño y en tiempo de ejecución de los resultados. Se han aplicado los comandos adecuados para observar la estructura interna y la inclusión de los elementos de depurado o *debugging*.

Como resultado final podemos decir que se ha afianzado la teoría aprendida sobre librerías a la vez que hemos practicado el desarrollo de software a bajo nivel, además de encontrar más información sobre herramientas de análisis (`binutils`) y líneas de aprendizaje sobre `make` y otros.