Tema 3. Compilación y Enlazado de Programas

Índice

Bibliografía básica

[Prieto 06] A. Prieto, A. Lloris, J.C. Torres. Introducción a la Informática. 4ª Edición. McGraw-Hill, 2006.

[Carretero 2007] Jesús Carretero et al., *Sistemas Operativos. Una visión aplicada*, McGraw-Hill. 2007. Capítulo 5: Gestión de memoria.

[Levine 2000] John R. Levine, *Linkers & loader*, Morgan Kaufman, 2000. Disponible en la dirección: http://www.iecc.com/linker/.

[Beazley 2001] D. Beazly, B.D. Ward, y I.R. Cooke, "The Inside Story on Shared Libraries and Dynamic Loading", *Computing in Science & Engineering*, páginas 90-97. September-october, 2001.

[Aho 2008] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compiladores. Principios, Técnicas y Herramientas (2ª Edición). Addison Wesley, 2008.

1. Introducción

En el tema anterior, hemos visto como una de las responsabilidades del sistema operativo es la gestión de memoria. Sin embargo, como veremos en éste tema, la gestión de memoria no es solo responsabilidad del sistema operativo, sino que ésta se realiza en colaboración con los lenguajes de programación, el compilador, el montador y el hardware de gestión de memoria (MMU – *Memory Management Unit*, componente de la CPU que realiza las labores hardware de gestión de memoria, por ejemplo, realizada la paginación y/o segmentación). Todos estos componentes se reparten la tarea de gestionar la memoria, cada uno con diferentes responsabilidades. El objetivo de este tema es conocer qué hacen y cómo se integran los diversos componentes para proporcionar la funcionalidad que se necesita.

En la gestión de memoria se puede distinguir tres niveles:

- *Nivel de procesos* – establece cómo se reparte la memoria del sistema entre los procesos existentes, y es gestionado por el sistema operativo.
- *Nivel de regiones* – establece cómo se distribuye el espacio asignado a un proceso entre las diferentes regiones del mismo. También está gestionado por el SO.
- *Nivel de zonas* – determina cómo se reparte el espacio de una región entre las distintas zonas de la misma. Si bien la mayoría de las regiones son homogéneas, existen regiones, como el heap y la pila, que mantienen zonas diferentes en su interior. Este nivel está gestionado por el lenguaje de programación pero con cierto soporte del SO. Para gestionar esta zona se suministran funciones como:
  - Reservar una zona, como hacer el operador new en C++
  - Liberar una zona reservada, como hace delete de C++
  - Cambiar el tamaño de una zona reservada, por ejemplo, operación realloc de C.

Desde el punto de vista de la gestión de memoria, las necesidades de los procesos son:

1. Tener un espacio lógico independiente. En un SO multiprogramado no se conoce a priori la posición de memoria que ocupará un proceso cuando se cargue para su ejecución. Por tanto, es evidente que ni en la fase de programación, ni en la compilación-montaje, debe haber referencias a las direcciones de memoria reales donde se ejecutará el programa. En consecuencia, las referencias a memoria del código máquina corresponderán a direcciones de memoria dentro del ámbito del programa (normalmente en un rango de 0 al máximo posible de direccionamiento) y no tendrán relación con las direcciones de RAM donde se cargue el programa. Por todo ello, en sistemas multiprogramados es necesario realizar un proceso de traducción (reubicación) de las direcciones de memoria a las que hace referencia el programa (**direcciones lógicas**) a las direcciones de memoria principal asignadas (**direcciones físicas**). La reubicación puede realizarse por el software (reubicación software) antes de iniciar la ejecución del programa (como veremos en este tema) o bien puede ser reubicación hardware, en tiempo de ejecución, como vimos en el Tema 2 al hablar de paginación y segmentación.
2. Estar protegido de otros programas que se estén ejecutando simultáneamente. La gestión de memoria realizada por el SO garantiza que un proceso no pueda afectar a otros, por ejemplo, que un desbordamiento de pila o error de direccionamiento pueda acceder a memoria que no le pertenece. La protección está integrada en los mecanismos de traducción de direcciones.
3. Compartir memoria con otros procesos para comunicarse. Aunque el SO aísle los procesos por protección debe permitir que estos compartan memoria para poder intercambiar información. Esto se puede realizar permitiendo que dos direcciones lógicas de dos procesos accedan a la misma dirección física. Esto puede causar problemas de autorreferencias como veremos en el apartado de bibliotecas.
4. Tener soporte a las diferentes regiones. Como veremos el mapa de memoria de un proceso no es ni homogéneo ni estático, por lo que necesitamos soporte del sistema de gestión de memoria para poder manipular las diferentes regiones según sus propiedades.
5. Tener facilidades para la depuración. La construcción de un programa no está exenta de errores. Algunos errores pueden detectarse en tiempo de compilación o enlace. Otros solo se detecta en ejecución. La solución a algunos de los errores del primer tipo, la veremos en este tema. Respecto a

los segundos, indicar que en el Tema 4 veremos cómo usar un depurador para solventarlos. El tema de depuración debe estar contemplado en el de gestión de memoria por que el depurador es un programa que accede al mapa de memoria del proceso depurado para consultar variables, instrucción ejecutada, etc. y por tanto es necesario contemplar que se rebaje el nivel de protección del proceso depurado.

6. Poder usar un mapa de memoria muy grande si lo necesita. Para eliminar restricciones de memoria de cada a la programación de aplicaciones, el mapa de memoria de un proceso debe ser lo más grande posible. Es responsabilidad del SO ofrecer el máximo espacio a los programas. Este punto lo abordaremos en la asignatura de Sistemas Operativos.
7. Utilizar diferentes tipos de objetos de memoria. Como indicamos en la lista de necesidades, no todos los datos que utiliza una aplicación tienen las mismas características. Por un lado, hay datos constantes que no deben ser modificados por ninguna sentencia del programa. Dentro de los datos que pueden variar, se presentan varias alternativas: (a) datos estáticos, existen durante toda la ejecución del programa y se conocen desde el inicio del programa por lo que se les habilita espacio de almacenamiento, que se mantiene durante toda la ejecución; (b) datos dinámicos asociados a la ejecución de una función cuya su vida está asociada a la activación de una función, creándose en la invocación de la misma y destruyéndose cuando esta termina; (c) datos dinámicos controlados por el programa cuyo tiempo de vida no está vinculado a la activación de una función sino bajo el control directo del programa que los crea cuando los necesita. El espacio que ocupan se libera cuando no es necesario. Este tipo de datos se almacena en una región denominada *heap*.
8. Poder hacer persistentes los datos que así lo requieran, es decir, poder almacenarlos en archivos.
9. Desarrollarse de una forma modular. Cuando se afronta el desarrollo de una aplicación de cierta envergadura, es conveniente hacerlo de una forma modular. De momento entenderemos por módulo una unidad de compilación independiente. El desarrollo modular permite programar y compilar cada módulo de forma independiente, para finalmente integrar todos los módulos y conformar el ejecutable final. Esta técnica facilita el desarrollo incremental y fomenta la reutilización de código. Como podemos observar, el código objeto resultante de la compilación de un módulo sólo posee referencias a direcciones de memoria dentro del ámbito del mismo. Por tanto, será necesario realizar una reubicación de módulos para obtener el programa final. Esta labor es responsabilidad exclusiva del sistema de compilación.
10. Poder realizar una carga dinámica de módulos. Cada vez más aplicaciones requieren poder añadir nueva funcionalidad sin necesidad de volver a compilarlas, o incluso, de volver a arrancarlas (como ocurre con los *plugins*). Como veremos esta necesidad puede satisfacerse con el enlazado explícito de bibliotecas dinámicas.

2. Modelo de memoria de un proceso

En este apartado, nos centraremos en aspectos relacionados con la gestión del mapa de memoria de un proceso, incluyendo el proceso que va desde la generación del ejecutable hasta la carga del mismo en memoria, es decir, la gestión de los niveles de región y zona, que citábamos antes.

Primero, analizaremos cómo se implementan los diferentes tipos de objetos de memoria requeridos por el programa y cuál es su correspondencia con el mapa de memoria del proceso. A continuación, se estudia cual es el ciclo de vida de un programa desde que tenemos el archivo del código fuente hasta su ejecución, pasando por la compilación del mismo. Como parte de este estudio, mostraremos como se reparten las distintas etapas de reubicación, identificadas en el apartado anterior, entre las fases del ciclo de vida. Veremos también cual es la estructura de un archivo ejecutable y cómo se gestionan las bibliotecas.

Volvamos a analizar cada uno de los tipos de datos identificados con anterioridad (punto 7 del apartado anterior) desde el punto de vista de su implementación:

- **Datos estáticos** - pueden ser de distinto tipo:
  - *Globales* a todo el programa, módulo o locales a una función, dependiendo del ámbito de visibilidad de la variable.

- *Constantes* o *variables* – se deberá asegurar que no se modifican. El compilador puede hacer esta comprobación y generar un error en caso de que se intente modificar. Pero sería necesario volver asegurarse en tiempo de ejecución de que no se intentan modificar, por lo cual situaremos este tipo de dato en una región que no pueda modificarse.

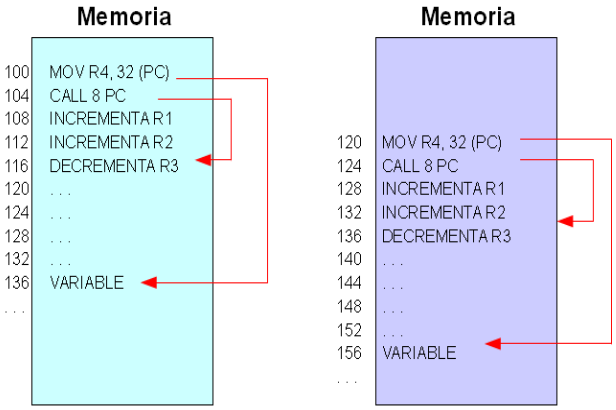


Figura 1 Ejemplo de código PIC

- *Con* o *sin valor inicial* asociado – si tiene un valor asociado, deberemos asegurarnos de que este cargado en la memoria cuando se vaya a usar. Como el mapa inicial del proceso se construye a partir del ejecutable, habrá que almacenar ese valor en el mismo. En cuanto a su implementación, es habitual utilizar usar direccionamiento absoluto por ser más eficiente. Pero en ocasiones es más interesante utilizar un direccionamiento relativo a un registro. Si bien es ligeramente menos eficiente, nos permite generar **código independiente de la posición** (PIC). Decimos que cierto código cumple esta propiedad si puede ejecutarse en cualquier parte de la memoria, de ahí el nombre. O visto desde otra forma, que no requiere reubicación. Para conseguir este tipo de código, todas las referencias a datos e instrucciones del mismo no deben referirse a direcciones concretas, sino de forma relativa, basándose en el valor almacenado en un registro. Como veremos, este tipo de código se utiliza en bibliotecas dinámicas.

- *Datos dinámicos asociados a la ejecución de una función* - este tipo de objetos se corresponden con las variables locales y los parámetros de una función. Como se crean en la invocación de la función, no tienen espacio inicial asignado en el mapa del proceso. Habitualmente, se almacena dinámicamente en la pila del proceso, en lo que se denomina **registro de activación**. Un registro de activación es una estructura que almacena, además de la variables locales y los parámetros, la información necesaria para retornar al punto de llamada de la función cuando finalice la misma. Se trata de un caso de nivel de zona en el que la petición de reserva de memoria se corresponde con la creación de un registro de activación y la liberación esta asociada a la eliminación de un registro de activación. La gestión de la pila esta realizada en parte por el SO. La dirección de este tipos de variables se determina en ejecución, ya que depende de la secuencia de llamadas que realice el proceso. Para acceder a una variable de este tipo hay que consultar el último registro de activación apilado pues es el que está asociado con la ejecución actual de la función. En el acceso se utiliza direccionamiento relativo al puntero de pila. Esto hace que no sea necesario reubicar este tipo de objetos y cumplen la propiedad PIC. El compilador resuelve todos los aspectos de implementación de este tipo de objetos. A diferencia de los datos anteriores, es el propio código ensamblador del programa (generado en la compilación) el encargado de realizar explícitamente la iniciación del mismo. Por ejemplo, sea el fragmento de programa en C:

```
#include <stdio.h>
int a();            // Declaramos tres funciones: a, b y c
int b();
int c();

int a()             //La función a invoca a las funciones b y c
{
   b();
   c();
   return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }
```

```
int main()          // programa principal
{
   a();
   return 0;
}
```

La Figura 2 muestra la evolución de la pila durante la ejecución del programa. Al final, ésta estará en equilibrio. Como se observa en cada invocación a función se apila en la misma un registro de activación o marco (frame).

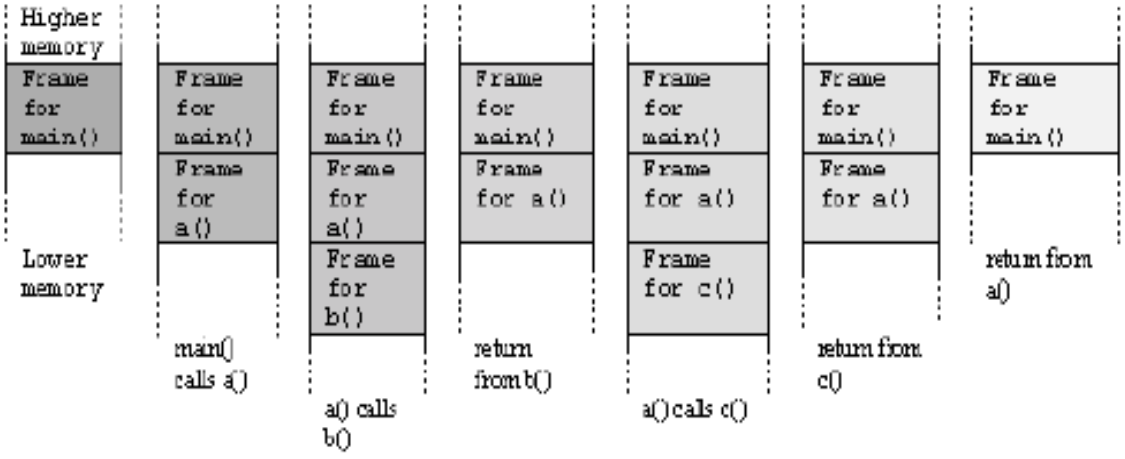


Figura 2.- Llamadas a función y registros de activación.

- **Datos dinámicos del heap** - estos datos los crea el programa cuando los necesita y los destruye cuando no le son necesarios. La liberación puede hacerla explícitamente el programa o se lleva a cabo mediante un mecanismo automático denominado *recolección de basura*. La dirección asignada a cada dato solo se conoce en tiempo de ejecución, cuando el programa asigna el heap. Por lo que es necesario un mecanismo de direccionamiento indirecto para acceder al dato: la posición de la memoria asignada a la variable asociada al dato no contiene el propio dato (como ocurre en los tipos anteriores) sino la dirección de memoria donde está almacenado el datos (contiene un *puntero*). Por tanto, no es necesario reubicarlos y cumplen la propiedad PIC. Debido a lo cual, el compilador y las bibliotecas del lenguaje resuelven todos los aspectos de implementación requeridos por este tipo de objetos con cierto apoyo del SO. La labor del SO en la gestión del *heap* se hace que sea mínima para: (a) minimiza las llamadas al sistema y mejora la eficiencia; (b) por flexibilidad, ya que cada lenguaje tiene su propio modelo de memoria dinámica y sería difícil de implementar por parte del SO una solución adecuada para todos los procesos. Para finalizar, indicar que algunas bibliotecas de gestión del *heap* disponen de versiones específicas para la fase de depuración de una aplicación e implementan técnicas que intentar detectar algunos errores que se pueden producir (punteros sueltos, desbordamientos, goteras) en tiempo de ejecución. Estas técnicas suponen una sobrecarga en tiempo y espacio pero son tolerables mientras estamos depurando el programa.

Programa 1.- Distintos tipos de objetos de memoria.

```
int a, b=7;                /* variables estáticas globales sin/con valor inicial*/
static int c, d=10;        /* variables estáticas de módulo sin/con valor inicial*/
const int e=11;            /* constante estática globales */
statit const int f=1       /* constante estática de módulo */
extern int g;              /* referencia a valor gobal de otro módulo */

void funcion(int h)        /* parámetro: variable dinámica de función */
{
       int i, j=8;         /* vars. dinámicas de función sin/con valor inicial */
       static int k, l=2;  /* vars. Estáticas locales sin/con valor */
       {
              int m, n=6;  /* vars. Dinámicas de bloque sin/con valor inicial */
       }
}
```

Programa 2.- Programa que utiliza los diferentes tipos de objetos de Programa 1.

```
struct tipo {
       int a, b;
};

int main(int argc, char *argv[])
{
       static struct tipo var_estatica;
       struct tipo var_dinamica;
       struct tipo *var_heap;
       var_heap= malloc(sizeof(struct tipo);
       var_estatica.b= 12;        /* 1 acceso con direccionamiento absoluto */
       var_dinanica.b= 14;        /* 1 acceso con direccionamiento relativo a SP */
       var_heap->b= 22;           /* 2 accesos con direccionamiento indirecto */
       return 0;
}
```

3. Ciclo de vida de un programa

Una vez que el programador ha finalizado la escritura del programa, éste debe pasar por varias fases antes de que pueda ejecutarse. Estas son:

1. *Preprocesado*: primera fase de la compilación en la que se procesan los includes, las instrucciones de compilación condicional, y las macros.
2. *Compilación*: segunda fase, en la que se toma como entrada el archivo generado por el preprocesador y el código fuente y se genera el código ensamblador.

File extension	Description
file_name.c	C source code which must be preprocessed.
file_name.i	C source code which should not be preprocessed.
file_name.ii	C++ source code which should not be preprocessed.
file_name.h	C header file (not to be compiled or linked).
file_name.cc file_name.cp file_name.cxx file_name.cpp file_name.c++ file_name.C	C++ source code which must be preprocessed. For file_name.cxx, the xx must both be literally character x and file_name.C, is capital c.
file_name.s	Assembler code.
file_name.S	Assembler code which must be preprocessed.
file_name.o	Object file by default, the object file name for a source file is made by replacing the extension .c, .i, .s etc with .o

Tabla 1.- Extensiones de archivos

3. *Ensamblado*: se convierte el código ensamblador en el archivo objeto.
4. *Enlazado o montaje*: es la fase final de la compilación, en la que se toman uno a más archivos objeto y las bibliotecas como entrada y se combinan para generar un único archivo ejecutable. En esta fase, se resuelven las referencias de símbolos externos, se asignan las direcciones finales a procedimientos/funciones y variables, y se revisa el código para reflejar las nuevas direcciones (reubicación).
5. *Carga y ejecución*: Como veremos en breve, el montador no completa la secuencia de reubicaciones requeridas para ejecutar el programa, por lo que será necesario que en la etapa de carga del programa en memoria o durante la ejecución del mismo se complete esta secuencia.

La Figura 3 muestra la estructura general del proceso, y la Tabla 1 indica que archivos intermedios se generan al final de cada fase. Vamos primero a describir con más detalle cada una de estas etapas. En el Apartado 3.7, las ilustraremos usando un ejemplo para ver como se materializa cada etapa en un sistema GNU/Linux.

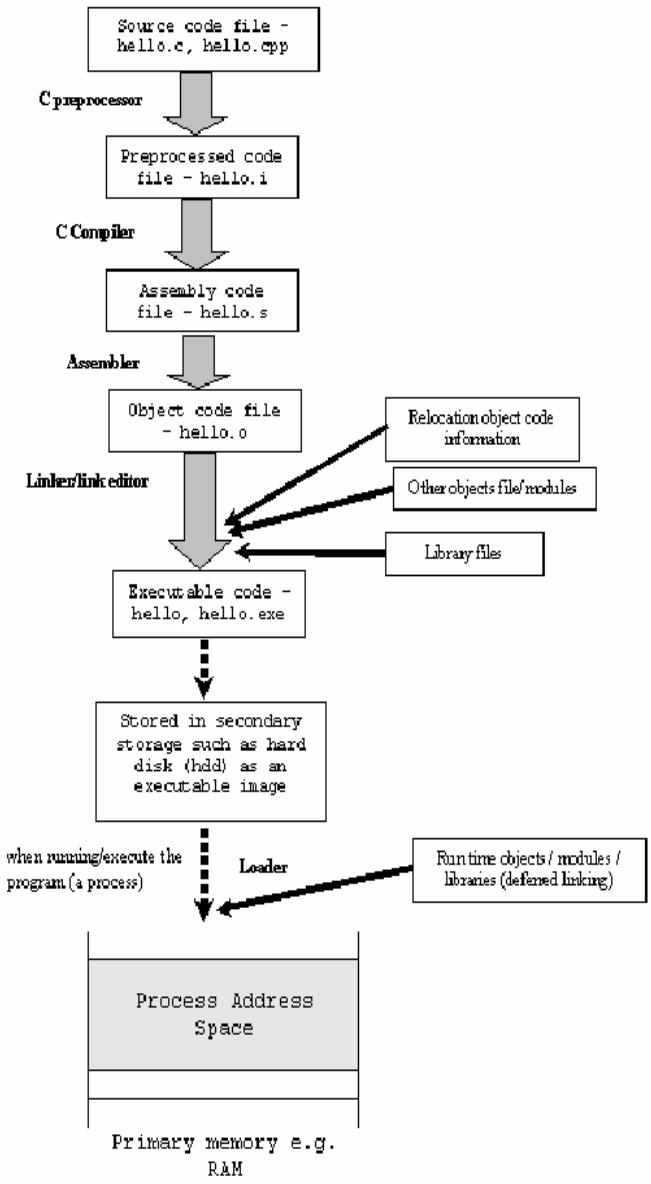


Figura 3.- Fases del procesamiento de un programa y archivos involucrados.

3.1 Compilación

3.1.1 Proceso de compilación

Habitualmente los programadores desarrollan sus aplicaciones utilizando lenguajes de alto nivel, dividiendo la funcionalidad del mismo en varios módulos (archivos de código fuente) para facilitar el desarrollo incremental y la reutilización de código. El compilador procesa cada uno de los archivos de código fuente y genera su archivo objeto correspondiente. Se encarga de realizar las siguientes acciones:

- Genera el código objeto y determina cuanto espacio ocupan los diferentes tipos de datos. El compilador organiza toda la información en secciones, que son unidades de organización (ver el apartado 3.5 "Formato de un objeto/ejecutable").
- Asigna direcciones a los símbolos estáticos definidos en el módulo. Se asignan direcciones consecutivas: primero, a las constantes; luego, a variables con valor inicial, finalmente, a las variables sin valor inicial.
- Resuelve las referencias a los símbolos estáticos definidos en el módulo, ya sean instrucciones o datos. Esta referencias pueden resolverse con un direccionamiento absoluto, que habrá que reubicar, o relativo al PC, que favorece la generación de código PIC, aunque también puede requerir reubicación.
- Respecto a las referencias a símbolos estáticos definidos en otros módulos de la aplicación, habrá que resolverlas en la fase de enlazado.
- Las referencias a símbolos dinámicos se resuelven con direccionamiento relativo a pila, si los datos están asociados a la invocación de una función, o direccionamiento indirecto con punteros en el caso de variables del *heap*. Este tipo de variables no aparecen en el archivo objeto y no necesitan reubicación.

Además de estas secciones, el compilador genera otras necesarias para realizar su labor y facilitar el trabajo del enlazador:

- El enlazador necesita una sección que almacene información sobre las instrucciones que necesitan reubicación o están pendientes de resolver su referencia a un símbolo externo.
- Otra sección almacena la **tabla de símbolos**, que recoge los símbolo globales definidos en el módulo (que se exportarán) y a qué símbolos externos se hace referencia (se importaran).
- Si se compila con la opción de depuración, se genera una sección con **información de depuración** para permitir al depurador en tiempo de ejecutar saber a qué línea de código corresponde una instrucción o a qué variable corresponde una dirección de memoria.

Como resultado de la compilación se genera un archivo objeto similar al que muestra la Figura 13. Existen diferentes formatos para este archivo: ELF, a.out, etc., que estudiaremos con más detalle en el Apartado 3.4.

Para verlo con un ejemplo vamos a construir un programa en C muy sencillo, que es una variante del clásico "Hola Mundo".

```
#include <stdio.h>
int x = 42;

int main()
{
      printf("Hola Mundo, x = %d\n", x);
}
```

Si compilamos este programa, se produce un archivo objeto que contiene una s*ección de texto* con las instrucciones máquina del programa, una *sección de datos* con la variable global x, y una sección de solo lectura con la cadena "Hola Mundo, x = %d\n". Además, el archivo objeto contiene una tabla de símbolos para todos los identificadores que aparecen en código fuente. Podemos ver estos identificadores con la orden Unix `nm`:

```
$ gcc -c hola.c
$ nm hola.o
00000000 T main
         U printf
00000000 D x
```

La tabla de símbolos almacena la posición relativa al inicio de su respectiva sección del símbolo en cuestión. Como ocurre con la `x` y `main`, que son respectivamente la primera variable y la primera función, por lo que tiene el desplazamiento 0. Otros símbolos, como `printf`, se marcan con una U indicando que si bien se usan en el programa, no están definidos. También, podemos ver los símbolos, incluidos los de depuración, con la opción `-a` (`nm -a hola.o`).

3.1.2 Qué es y cómo funciona un compilador

Un traductor es un programa que recibe como entrada un texto en un lenguaje de programación concreto y produce, como salida, un texto en lenguaje máquina equivalente. El programa inicial se denomina *programa fuente* y el programa obtenido, *programa objeto*. Las Figuras 4 y 5 muestran el proceso de traducción tanto si se trata de un compilador (por ejemplo, g++) como de un intérprete (p.e. Bash shell).

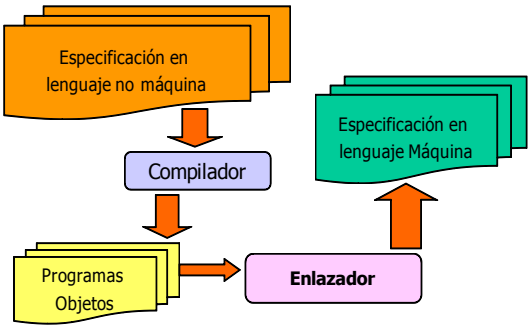


Figura 4*:* Proceso de traducción: Compilador.

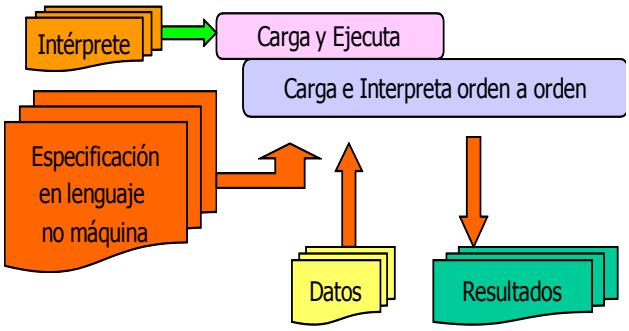


Figura 5: Proceso de traducción: Intérprete.

Compilador: Traduce la especificación de entrada a lenguaje máquina incompleto y con instrucciones máquina incompletas.

Enlazador: Enlaza los programas objetos y completa las instrucciones máquina incompletas, generando un ejecutable.

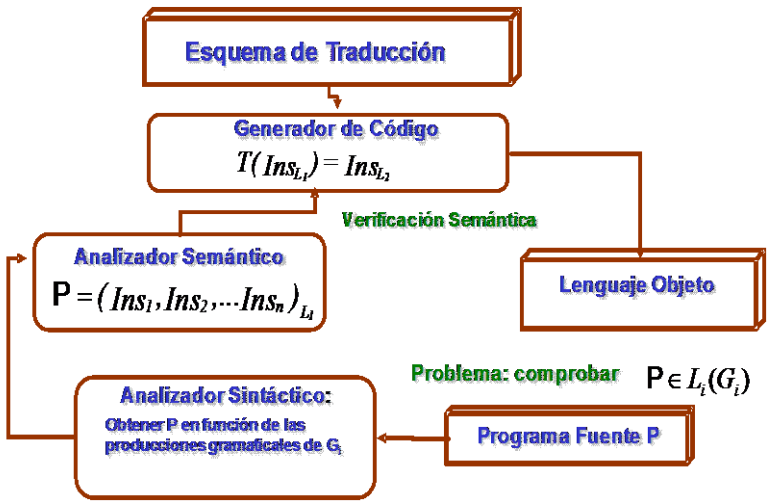


Figura 6: Esquema de traducción.

En todo proceso de traducción, la fase principal viene definida por el esquema de traducción. Se puede definir como el proceso por el cual a una frase del lenguaje fuente que es correcta a nivel léxico, sintáctico y semántico, se le asocia un conjunto de frases correspondientes al lenguaje objeto. En la figura 4 se muestra el citado esquema de traducción. Como puede apreciarse, existen dos lados en lo que sería el esquema de traducción, también conocido como generador de código. A la izquierda habría que resolver el problema de la verificación sintáctica y la semántica a partir de un programa fuente, mientras que a la derecha se obtendría el programa objeto.

La complejidad del proceso de la verificación sintáctica va a depender del tipo de gramática que define el lenguaje fuente.

Una gramática es una especificación para la estructura sintáctica de un lenguaje y se define mediante la cuádrupla (N,T,P,S) donde:

- N representa el conjunto de símbolos no terminales.
- T es el conjunto de símbolos terminales.
- P es el conjunto de producciones (o reglas) gramaticales.
- S es el símbolo (no terminal) inicial de la gramática.

3.1.2.1. Fases de Traducción

[Aho08] (pp. 4-12)

Para facilitar el uso de los computadores se han desarrollado lenguajes de programación que permiten utilizar una simbología y una terminología próximas a las utilizadas tradicionalmente en la descripción de problemas y una terminología próximas a las utilizadas tradicionalmente en la descripción de problemas.

Un compilador es una caja simple que a partir de un programa fuente genera su equivalencia semántica a un programa destino. Si abrimos esta caja un poco, podremos ver que hay dos procesos en esta asignación: análisis y síntesis (ver Figura 7).

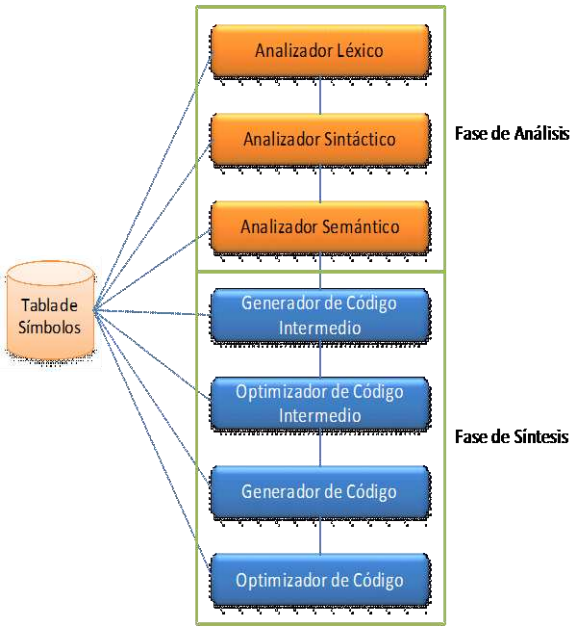


Figura 7: Fases de un compilador

La parte de *análisis* divide el programa fuente en componentes e impone una estructura gramatical sobre ellos. Después utiliza esta estructura para crear una representación intermedia del programa fuente. Si la parte del análisis detecta que el programa fuente está mal formado en cuanto a los componentes y/o la sintaxis, o que no tiene una semántica consistente, entonces debe proporcionar mensajes informativos para que el usuario pueda corregirlos. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada *tabla de símbolos*, la cual se pasa junto con la representación intermedia a la parte de la síntesis.

La parte de *síntesis* construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos. A la parte del análisis se le llama comúnmente el *front-end* del compilador; la parte de síntesis (propiamente la traducción) es el *back-end*.

Si examinamos el proceso de compilación con más detalle, podremos ver que opera como una secuencia de fases, cada una de las cuales transforma una representación del programa fuente en otro. En la práctica, varias fases pueden agruparse y las representaciones intermedias entre las fases agrupadas no necesitan construirse de manera explícita. La tabla de símbolos, que almacena información sobre todo el programa fuente, se utiliza en todas las fases del compilador. Cabe destacar que la fase de optimización de código es opcional, dicho lo cual, podría eliminarse de esta estructura de fases del compilador.

A) Análisis de Léxico [Aho08] (Cap.3, pp.109-145)

Al ser la primera fase de un compilador, la principal tarea del analizador léxico es leer los caracteres de la entrada del programa fuente, agruparlos en lexemas (palabras) y producir como salida un secuencia de tokens para cada lexema en el programa fuente. Los conceptos subyacentes en esta etapa son los siguientes:

Lexema o **Palabra**: Es un conjunto de caracteres del alfabeto que tienen significado propio.

Token: Es el concepto asociado a un conjunto de lexemas o palabras que, según la gramática del lenguaje fuente, tienen la misma misión sintáctica. Por ejemplo, dada la gramática de un lenguaje natural, los artículos determinados e indeterminados son lexemas o palabras que pueden aparecer en el mismo lugar de una frase.

Aquellos tokens formados por más de un lexema, deben disponer de un valor de atributo para que, en fases posteriores, pueda distinguirse el lexema reconocido inicialmente en esta fase de cara a las fases posteriores (por ejemplo, dado un token que recoja los operadores de multiplicación y división, en la fase del semántico en adelante se necesita identificar qué operador se recogió en el léxico para asignar el código de operación, suponiendo que se traduce a lenguaje máquina).

Patrón: Es una descripción de la forma que pueden tomar los lexemas de un token. En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres que forman dicha palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja que se relaciona mediante muchas cadenas.

Obtenido el flujo de tokens, cada uno de estos se envían al analizador sintáctico para su posterior análisis. En la figura 6 se puede apreciar el esquema de interacción entre el analizador léxico y el sintáctico.

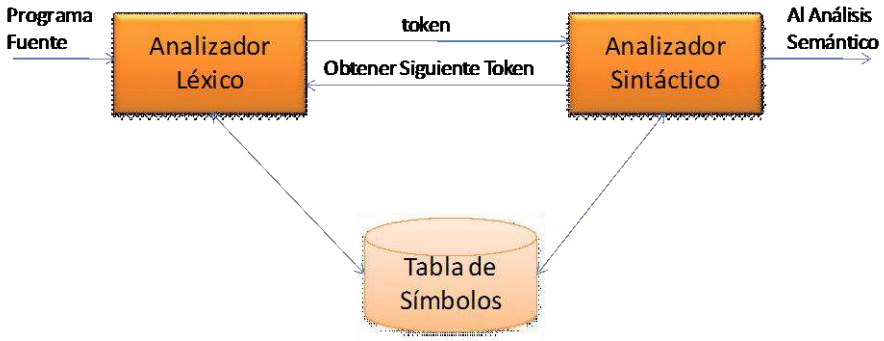


Figura 8: Interacción entre el analizador léxico y el sintáctico.

Por lo tanto, el analizador léxico es la parte del compilador que lee el texto de origen, identifica lexemas, les asocia el token al que pertenecen pero, además, debe eliminar los comentarios y caracteres superfluos existentes en el texto de entrada (espacios en blanco, tabuladores y retornos de carro).

Existen varias razones por las cuales la parte correspondiente al análisis de un compilador se separa en fases de léxico y sintáctico:

1. La sencillez en el diseño es la consideración más importante. La separación del léxico y el sintáctico a menudo nos permite simplificar la etapa del sintáctico simplemente por el hecho de simplificar el texto de entrada eliminando espacios en blanco, comentarios y otros caracteres superfluos.
2. Se mejora la eficiencia del compilador. Un léxico separado nos permite aplicar técnicas especializadas que sirven sólo para la tarea léxica, no para el trabajo del análisis sintáctico.
3. Se mejora la portabilidad del compilador. Las peculiaridades específicas de los dispositivos de entrada pueden restringirse al analizador de léxico.

■

En muchos lenguajes de programación, las siguientes clases cubren la mayoría, si no es que todos, los

tokens:

1. Un token para cada palabra reservada. El patrón para una palabra reservada lo forman los caracteres de esa palabra clave.
2. Los tokens para los operadores, ya sea en forma individual o en grupos como el token comparación, mencionado en el ejemplo anterior.
3. Un token que representa a todos los identificadores, ya sean de variables o de subprogramas.
4. Uno o más tokens que representan a las constantes, como los números y las cadenas de literales.
5. Tokens para cada signo de puntuación, como los paréntesis izquierdo y derecho, la coma, punto y coma, corchetes, llaves, …

Acontecerá un error léxico cuando el carácter de la entrada no tenga asociado a ninguno de los patrones disponibles en nuestra lista de tokens. Por ejemplo, si aparece un carácter extraño en la formación de un identificador, el analizador léxico deberá indicar que se ha producido un error de naturaleza léxica.

B) Análisis Sintáctico [Prie06] (Cap.14. pp.585-587)

La sintaxis de un lenguaje de programación indica cómo deben escribirse los programas. Se fundamentan en un conjunto de reglas de sintaxis también conocidas como gramática del lenguaje. Un programa será sintácticamente correcto cuando sus expresiones, sentencias declarativas, asignaciones, etc. estén bien formados, esto es, cada sentencia sea estructuralmente correcta. De este modo, en lenguaje C, es sintácticamente incorrecta la siguiente sentencia:

```
a+b=(c-d)*17;
```

La naturaleza del error sintáctico de la anterior sentencia reside en que antes del operador de asignación sólo ha de aparecer un identificador. El error se detectaría justo al leer el token operador binario (+). Ahí no se espera tal operador y sí un operador de asignación (=).

Se ha de advertir que una construcción puede ser sintácticamente correcta pero carecer de significado. Un ejemplo de sentencia que cumpliría la anterior afirmación sería cuando a una variable de tipo cadena de caracteres se le trata de asignar un valor entero.

C) Análisis Semántico [Prie06] (Cap.14. pp.587-588)

La semántica de un lenguaje de programación es el significado dado a las distintas construcciones sintácticas. El proceso de traducción es, en esencia, la generación de un código en lenguaje máquina con el mismo significado que el código fuente. En los lenguajes de programación, el significado está ligado a la estructura sintáctica de las sentencias. Así, una sentencia de asignación significa transferir el valor de la expresión de la derecha al identificador de la izquierda.

En el proceso de traducción, el significado de las sentencias se obtiene de la identificación sintáctica de las construcciones sintácticas y de la información almacenada en las tablas de símbolos.

Durante la fase de análisis semántico se pueden producir errores, cuando se detectan construcciones "sin un significado correcto". Por ejemplo, asignar a una variable definida como dato numérico en simple precisión el valor de una variable cadena de caracteres es semánticamente incorrecto para algunos compiladores.

D) Generación y Optimización de Código [Prie06] (Cap.14. pp.558)

En esta fase se crea un archivo con un código en lenguaje objeto (normalmente lenguaje máquina) con el mismo significado que en el texto fuente. El archivo-objeto generado puede ser (dependiendo del compilador) directamente ejecutable, o necesitar otros pasos previos a la ejecución tales como ensamblado, enlazado y carga. En algunas ocasiones se utiliza un lenguaje intermedio (distinto del código objeto final), con el propósito de facilitar la optimización de código.

En la generación de código intermedio se completa y se consulta la tabla de símbolos generada en las fases anteriores. También se realiza la asignación de memoria a los datos definidos en el programa.

La generación de código puede realizarse añadiendo procedimientos en determinados puntos del proceso de análisis que generen las instrucciones en lenguaje objeto equivalentes a cada construcción en lenguaje

fuente reconocida.

En la fase de optimización se mejora el código intermedio analizándose el programa objeto en su totalidad. Un programa puede incluir dentro de un bucle una sentencia que asigna a una variable un valor constante sin alterar dicho valor a lo largo del bucle. Si esa sentencia se deja dentro de bucle se ejecutaría tantas veces como se iterase el mismo. En esta ocasión, sería semánticamente equivalente si dicha sentencia se sacara del bucle situándola antes del mismo con el correspondiente ahorro de ejecución superflua. Todo ello reduce el tiempo de ejecución.

3.1.2.2 Intérpretes [Prie06] (Cap.14. pp.589-590)

Un intérprete hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. El intérprete capta una sentencia fuente, la analiza e interpreta dando lugar a su ejecución inmediata, no creándose, por tanto, un archivo o programa objeto almacenable en memoria masiva para ulteriores ejecuciones. Por tanto, la ejecución del programa está supervisada por el intérprete.

En la práctica, el usuario crea un archivo con el programa fuente. Esto suele realizarse con un editor específico del propio intérprete del lenguaje. Según se van almacenando las instrucciones simbólicas, se analizan y se producen los mensajes de error correspondientes; así el usuario puede proceder inmediatamente a su corrección. Una vez creado el archivo fuente el usuario puede dar la orden de ejecución y el intérprete lo ejecuta línea a línea. Siempre el análisis antecede inmediatamente a la ejecución, de forma que:

a) Si una sentencia forma parte de un bucle, se analiza tantas veces como tenga que ejecutarse el bucle.

b) Las optimizaciones sólo se realizan dentro del contexto de cada sentencia y no contemplándose el programa o sus estructuras en conjunto. La optimización comentada en la sección anterior no se podría realizar en el caso de un intérprete.

c) Cada vez que utilicemos el programa tenemos que volver a analizarlo, ya que en la traducción no se genera un archivo objeto que poder guardar en memoria masiva (y utilizarlo en cada ejecución). Con un compilador, aunque la traducción sea más lenta, ésta sólo debe realizarse una vez (ya depurado el programa) y cuando deseamos ejecutar un programa ejecutamos el archivo objeto, que se tradujo previamente.

Los intérpretes, a pesar de los inconvenientes anteriores, son preferibles a los compiladores cuando el número de veces que se va a ejecutar el programa es muy bajo y no hay problemas de velocidad. Además, con ellos puede ser más fácil desarrollar programas. Esto es así porque normalmente la ejecución de un programa bajo un intérprete puede interrumpirse en cualquier momento para conocer los valores de las distintas variables y la instrucción fuente que acaba de ser ejecutada. Con un programa compilado no se podría realizar salvo que estuviera bajo el dominio de un depurador (debugger).

Obviamente, los intérpretes resultan más pedagógicos para aprender a programar ya que se pueden detectar y corregir más fácilmente sus errores.

3.2 Enlazado

Para generar el ejecutable, el enlazador debe agrupar los archivos objetos de la aplicación y bibliotecas, y resolver las referencias entre ellos (ver Figura 9). También deben realizar algunas de las reubicaciones necesarias dependiendo del esquema de gestión de memoria utilizado.

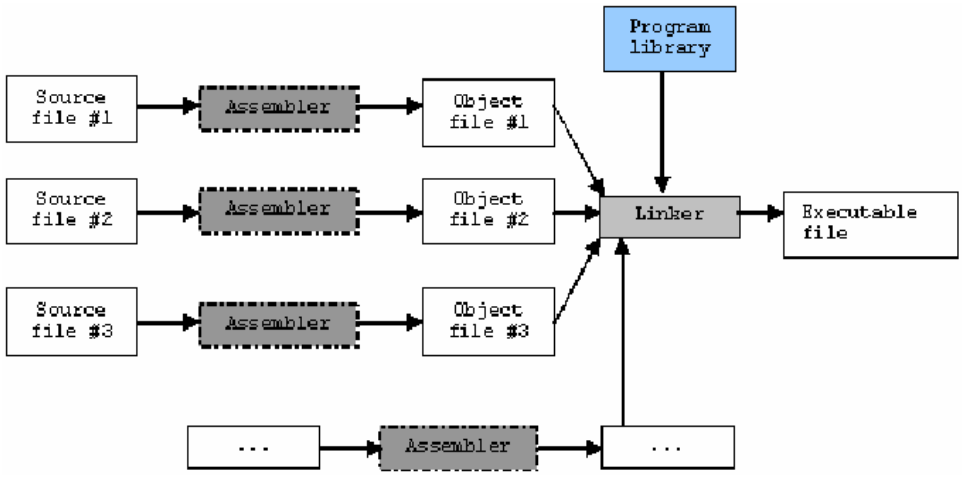


Figura 9.- Proceso de enlazado de archivos objeto.

Más concretamente, las labores que realiza el enlazador son:

1. Se completa la etapa de resolución de símbolos. Para ello, se buscan en las tablas de símbolos de los otros módulos los símbolos externos
2. Una vez resueltos los símbolos, se agrupan en regiones los módulos de las mismas características, de forma que cada región es la concatenación de secciones del mismo tipo (código, datos inicializados o no inicializados) presentes en cada módulo. Esto reduce el número de regiones que maneja el SO.
3. Llegados al punto anterior, hay que realizar **reubicación de módulos** , es decir, de sus referencias, necesaria para transformar las referencias dentro de los módulos a referencias dentro de las regiones. Ver Figura 10. Tras esta fase, cada archivo objeto tiene una *lista de reubicación* que contiene los nombres de los símbolos y los desplazamientos dentro del archivo que debe aún parchearse.

En este instante, sólo se ha realizado la reubicación de módulos, faltan aún la reubicación de regiones y la de procesos. En sistemas segmentados, la labor del enlazador finaliza aquí. En ellos se fusiona el nivel de procesos y regiones, siendo el ejecutable una mera concatenación de regiones (con la excepción de la región de datos no inicializados, que no se almacena), sin ninguna reubicación adicional. Se asigna un segmento a cada región. En los sistemas que no usan segmentación, la labor del enlazador prosigue con la etapa de reubicación de regiones (se convierte direcciones dentro de la región en direcciones dentro del contexto del mapa del proceso, Figura 11). Algunos compiladores realizan la reubicación de módulos y regiones a la vez por razones de eficiencia.

Aquí finaliza la labor del enlazador, la última etapa, reubicación del proceso, se deja para el momento de carga del ejecutable (reubicación software) o para cuando este se ejecuta (reubicación hardware). Esta última etapa sólo tiene sentido que la haga el enlazador en sistemas monoprogramados o en aquellos sistemas embebidos en los que los procesos siempre se ejecutan en las mismas direcciones físicas.

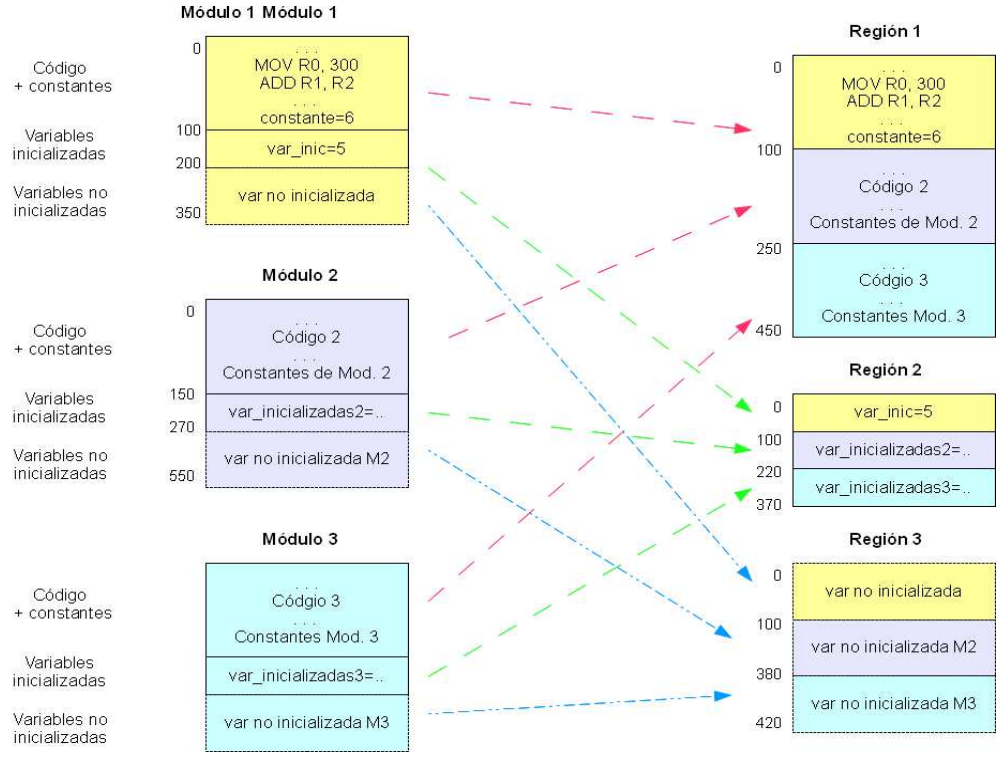


Figura 10.- Agrupamiento de módulos en regiones.

Volviendo al ejemplo, podemos ver la lista de reubicación con la orden `objdump`, que permite ver información de los archivos objeto, para ellos ejecutamos:

```
$ objdump -r hola.o        # -r imprime las entradas a reubicar
hola.o:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000b R_386_32          x
00000010 R_386_32          .rodata
0000001c R_386_PC32        printf
```

Como podemos ver, los símbolos a los que debemos asignar direcciones son: la variable `x`, la función `printf`, y la cadena de solo lectura `"Hola Mundo"`, que está en la sección `.rodata`.

Todos los identificadores tienen algunos de los tres atributos de enlazado (estrechamente relacionados con su ámbito): externo, interno, o sin enlazado. Estos atributos son propios para cada identificador en cada unidad de compilación y vienen determinados por la situación y el formato de la declaración en la que se introdujo cada uno, así como del uso explícito o implícito (por defecto) de los especificadores de tipo de almacenamiento static y *extern*.

El tipo de enlazado define una especie de ámbito, pues indica si el mismo nombre en otro ámbito se refiere al mismo objeto (variable o función) o a otro distinto:

- Cada instancia de un identificador con enlazado externo representa el mismo objeto o función a través del total de ficheros y librerías que componen el programa. Es el tipo de enlazado a utilizar con objetos cuyo identificador puede ser utilizado en unidades de compilación distinta de aquella en la que se ha definido. Por esta razón se dice que las etiquetas con enlazado externo son "globales" para el programa.

enlazado externo ↔ visibilidad global

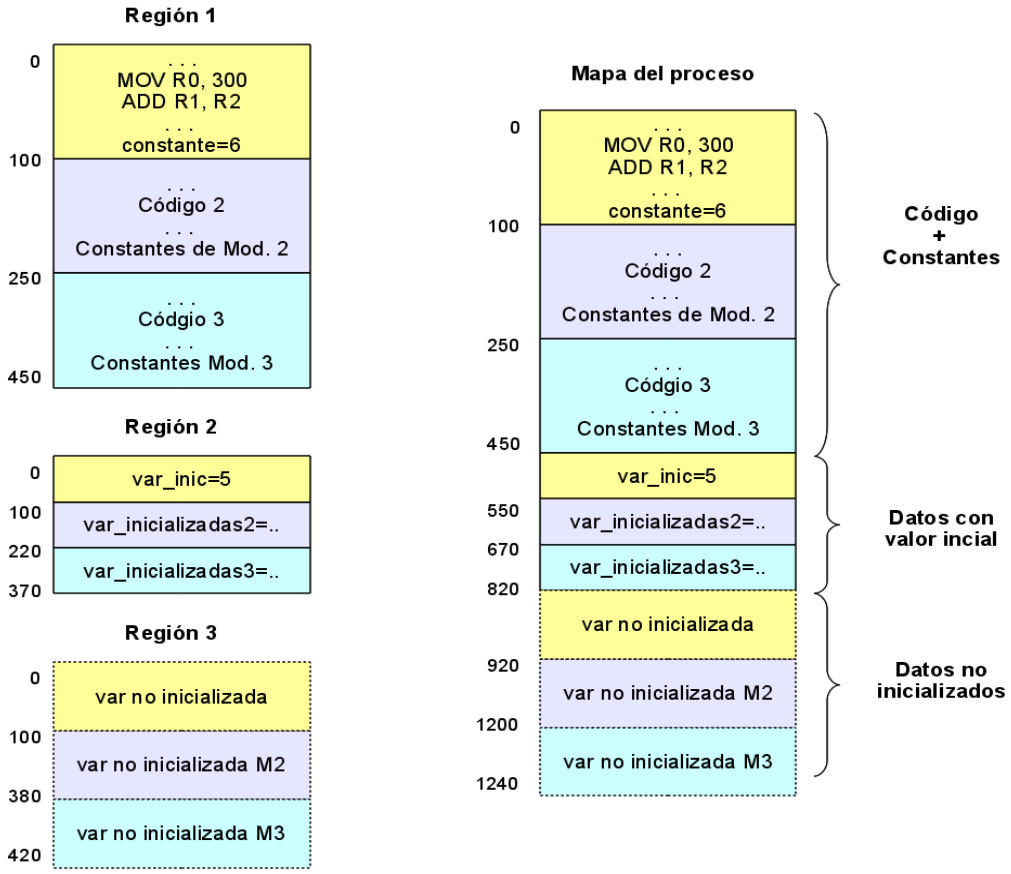


Figura 11.- Reubicación de regiones.

- Cada instancia de un identificador con enlazado interno representa el mismo objeto o función solo dentro del mismo fichero. Los objetos con el mismo nombre en otros ficheros son objetos distintos. Este tipo de objetos solo pueden utilizarse en la unidad de compilación en que se han definido, por lo que suele decirse que las etiquetas con enlazado interno son "locales" a sus unidades de compilación.

  enlazado interno ↔ visibilidad de fichero

- Las unidades sin enlazado representan entidades únicas. Por ejemplo, las variables declaradas dentro de un bloque, que no contengan el modificador **`extern`**, representan entidades únicas dentro del bloque, sin relación con nada en el exterior del mismo. Los objetos con el mismo nombre en otros bloques son objetos distintos. No obstante, es posible asignar punteros a este tipo de objetos sin enlazado, de forma que puedan ser accedidos desde cualquier punto del programa, incluso desde otras unidades de compilación.

  sin enlazado ↔ visibilidad de bloque.

Como puede verse, el hecho que un identificador utilizado en diversas unidades de compilación señale potencialmente a la misma entidad en todos los módulos, depende exclusivamente del tipo de enlazado que tenga en cada uno de ellos.

Para finalizar comentar las reglas de enlazado:

1ª En un fichero, cualquier objeto o identificador que tenga ámbito global deberá tener enlazado interno si su declaración contiene el especificador `static`.

2ª Si el mismo identificador aparece con ambos enlazados externo e interno, dentro del mismo fichero,

tendrá enlazado externo.

3ª Si en la declaración de un objeto o función aparece el especificador de tipo de almacenamiento extern, el identificador tiene el mismo enlazado que cualquier declaración visible del identificador con ámbito global. Si no existiera tal declaración visible, el identificador tiene enlazado externo.

4ª Si una función es declarada sin especificador de tipo de almacenamiento, su enlazado es el que correspondería si se hubiese utilizado extern (es decir, extern se supone por defecto en los prototipos de funciones).

5ª Si un objeto (que no sea una función) de ámbito global a un fichero es declarado sin especificar un tipo de almacenamiento, dicho identificador tendrá enlazado externo (ámbito de todo el programa). Como excepción, los objetos declarados const que no hayan sido declarados explícitamente extern tienen enlazado interno.

6ª Los identificadores que respondan a alguna de las condiciones que siguen tienen un atributo sin enlazado:

a. Cualquier identificador distinto de un objeto o una función (p.e., un identificador `typedef`).

b. Parámetros de funciones.

c. Identificadores para objetos de ámbito de bloque, entre corchetes { } , que sean declarados sin el especificador de clase extern.

Ejemplo:

```
int x;
static st = 0;

void func(int);

int main()
{
  for (x = 0; x < 10; x++) func(x);
}
void func(int j)
{
  st += j;
  cout << st << endl;
}
```

Comentario: Las etiqueta `x` tiene enlazado externo debido a su situación en el código, fuera de cualquier bloque o función. Podría ser utilizada desde cualquier otro módulo que la declarase a su vez como `extern` (regla 5). La etiqueta `func` tiene igualmente enlazado externo debido a su situación en el código (regla 4). La variable `j`, pertenece exclusivamente al ámbito de la función `func`, por lo que es sin enlazado (regla 6b). La variable `st` tiene enlazado interno. Debido a la declaración `static` solo es accesible desde dentro de su propio módulo (regla 1).

3.3 Carga y ejecución

Como indicábamos anteriormente, la reubicación se completa en la carga o ejecución. En cuyo caso, podemos distinguir entre tres casos, dependiendo del esquema de gestión de memoria utilizado:

- En segmentación, el enlazador sólo realiza reubicación de módulos. Se fusionan las dos etapas restantes. El cargador copia el programa en memoria sin modificarlo y es el hardware de gestión de memoria el encargado de realizar la reubicación combinada en tiempo de ejecución.
- En el resto de los casos, el enlazador ya ha realizado la reubicación de regiones. Si el hardware el capaz de reubicar los procesos, como ocurre con la paginación, el cargador copiará el programa en memoria sin modificarlo y es el hardware el que realiza la reubicación en ejecución.
- Si no se usa hardware de reubicación, ésta se realizará cuando el programa se cargue en memoria.

El cargador consulta la información necesaria que reside en el ejecutable y procede a la reubicación de procesos ajustando las referencias teniendo en cuenta donde se va a cargar el proceso. Como el caso anterior, pero es una reubicación software.

La Figura 12 muestra la secuencia de inicio de un programa que se invoca desde la llamada al sistema `execve()`, que carga un ejecutable en entornos Unix.

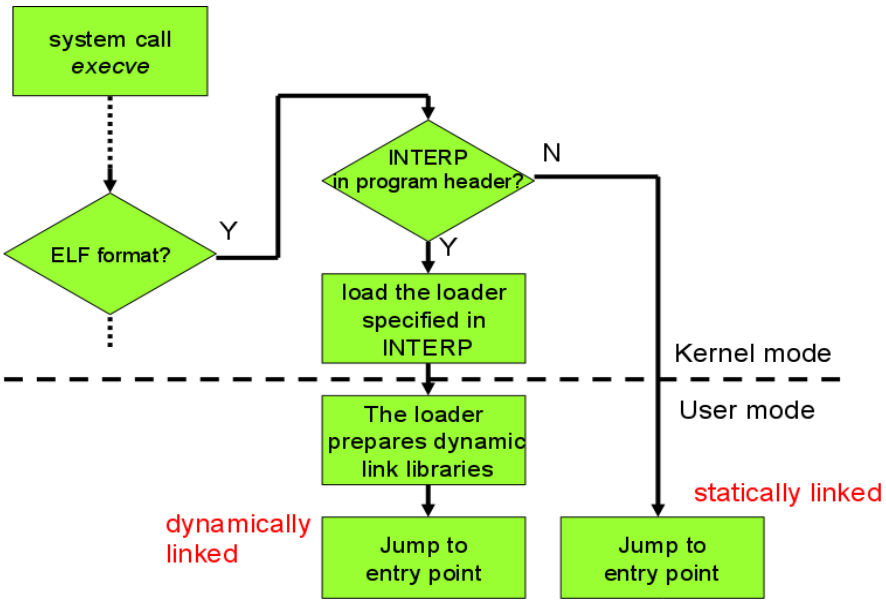


Figura 12.- Secuencia de inicio de la ejecución de un programa.

4. Formatos de archivos objetos y ejecutables

Como hemos visto, del resultado de la compilación se genera un archivo objeto, y como resultado del proceso de montaje se genera un archivo ejecutable, que contiene el código máquina del programa. El aspecto y características de estos archivos son similares. Las principales diferencias entre el objeto y el ejecutable son:

- Respecto del formato de la cabecera del ejecutable, como en el objeto, hay una tabla de regiones pero que contiene además el punto de inicio del ejecutable, es decir, la primera dirección que se cargará en el PC.
- En cuanto a las regiones, solo hay información de reubicación si esta se ha de realizar en la carga.

Existes diferentes formatos para estos archivos, en la Tabla 2 aparecen los más frecuentes, así como una breve descripción de los mismos.

Tabla 2.- Formatos de archivos objeto y ejecutables.

	Descripción
a.out	Es el formato original de los sistemas Unix. Consta de tres secciones: text, data y bss que se corresponden con el código, datos inicializados y sin inicializar. No tiene información para depuración.
COFF	El *Common Object File Format* posee múltiples secciones cada una con su cabecera pero están limitadas en número. Aunque permite información de depuración, ésta es limitada. Es el formato utilizado por Windows.
ELF	*Executable and Linking Format* es similar al COFF pero elimina algunas de sus restricciones. Se utiliza en los sistemas Unix modernos, incluido GNU/Linux y Solaris.

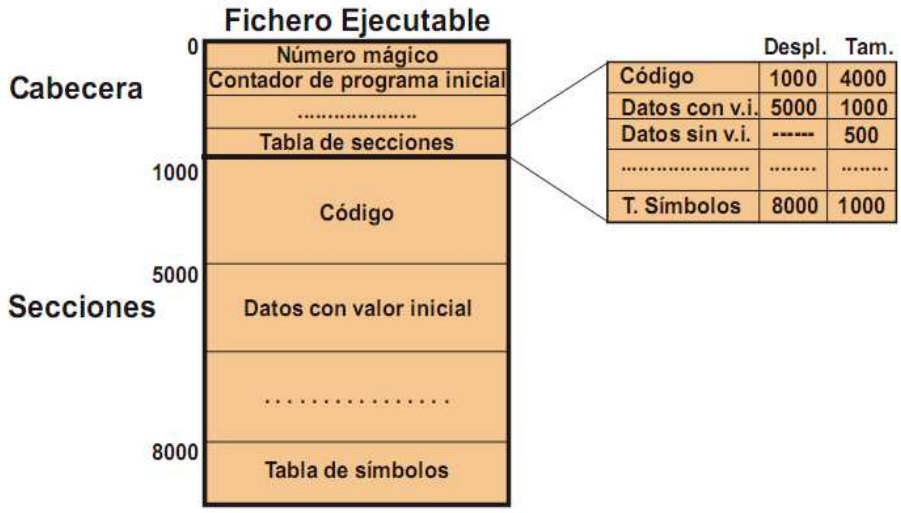


Figura 13.- Formato de un archivo ejecutable.

La Tabla 3 muestra las diferentes las secciones que son comunes a todos los formatos ejecutables aunque pueden recibir nombres diferentes según el compilador/enlazador.

Tabla 3.- Principales secciones de un ejecutable

Sección	Descripción
.text	Contiene las instrucciones del código ejecutable y es compartida entre todos los procesos que ejecutan el mismo binario. Esta región suele tener permisos de lectura y ejecución. Es una de las regiones más afectada por la optimización realizada por el compilador.
.bss	La región bss (*Block Started by Symbol*) contiene los datos no inicializados y variables estáticas. Dado que mantiene variables que no tienen aún valor asignado, no es necesario almacenar la imagen de esas variables. El tamaño de la bss que se requiere en ejecución se registra en el archivo objeto, pero no necesita espacio real en el objeto.
.data	Contiene las variables globales y estáticas inicializadas. Tiene permisos de lectura y escritura.
.rdata	También conocido como .rodata, contiene las constantes y cadenas literales.
.reloc	Almacena información necesaria para reubicar la imagen en la carga.
Tabla de símbolos	Como vimos, un símbolo es básicamente un nombre y su dirección. La tabla de símbolos almacena la información necesaria para localizar y reubicar definiciones y referencias simbólicas del programa. Es una matriz donde cada entrada contiene un símbolo.
Registros de reubicación	Es la información que utiliza el enlazador para ajustar los contenidos de las secciones que hay que reubicar.

- **ELF**

Un archivo con formato ELF describe tres tipos de archivos objeto:

- Un *archivo reubicable* que almacena código y datos que se puede enlazar con otros archivos objetos para crear un ejecutable o un archivo objeto compartido.
- Un *archivo ejecutable* que almacena un programa que puede ejecutarse; este archivo especifica

como la llamada al sistema `exec()` crea la imagen del proceso.

- Un *archivo objeto compartido* que almacena código y datos que puede ser enlazado en dos contextos. Primero, el editor de enlaces (`ld`) procesa el archivo objeto compartido con otros archivos objetos reubicables y compartidos para crear otro archivo objeto. Segundo, el enlazador dinámico lo combina con un ejecutable para crear la imagen de un proceso.

Las ordenes `objdump` y `readelf` nos permiten obtener información de un archivo ELF. Por ejemplo, podemos ver las secciones nuestro ejemplo:

```
$ objdump -h hola                #hemos suprimido algunas para simplificar

hola:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048154  08048154  00000154  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  . . .
 14 .text         0000018c  08048370  08048370  00000370  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .fini         0000001c  080484fc  080484fc  000004fc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 16 .rodata       0000001d  08048518  08048518  00000518  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .eh_frame_hdr 0000001c  08048538  08048538  00000538  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .eh_frame     0000005c  08048554  08048554  00000554  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 19 .ctors        00000008  08049f0c  08049f0c  00000f0c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 20 .dtors        00000008  08049f14  08049f14  00000f14  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 21 .jcr          00000004  08049f1c  08049f1c  00000f1c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 22 .dynamic      000000d0  08049f20  08049f20  00000f20  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 23 .got          00000004  08049ff0  08049ff0  00000ff0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 24 .got.plt      00000018  08049ff4  08049ff4  00000ff4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 25 .data         0000000c  0804a00c  0804a00c  0000100c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 26 .bss          00000008  0804a018  0804a018  00001018  2**2
                  ALLOC
 . . .
 29 .debug_aranges 00000040  00000000  00000000  00001180  2**3
                  CONTENTS, READONLY, DEBUGGING
 30 .debug_pubnames 0000005f  00000000  00000000  000011c0  2**0
                  CONTENTS, READONLY, DEBUGGING
 31 .debug_info   00000211  00000000  00000000  0000121f  2**0
                  CONTENTS, READONLY, DEBUGGING
 . . .
```

Comentaremos algunas de las secciones del ELF que no hemos descrito en la Tabla anterior:

`.interp` – programa interprete, por ejemplo, en Linux `/lib/ld-linux.so.2`.

`.ctors` – constructor, función requerida para la asignación de memoria para un objeto.

`.dtors` – destructores, función para liberar memoria cuando no se usa.

`.got` – Tabla de Desplazamientos Globales (la veremos al hablar de bibliotecas)

La Figura 14 muestra las dos vistas de un ELF: (a) la vista enlazada, y (b) la vista ejecutable, que se utilizará cuando se ejecute el programa, trata con segmentos (segmento = agrupación de secciones relacionadas, por ejemplo, `data` y `bss` se agrupan en el segmento data).

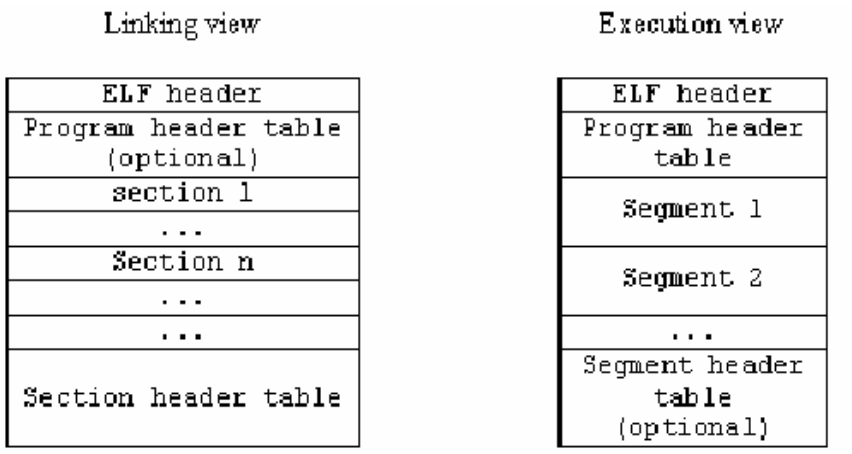


Figura 14.- ELF simplificado: vista enlazada y vista ejecutable.

Aquellas secciones del ejecutable que no son estrictamente necesarias para la ejecución de un programa (tabla de símbolos, información de depuración, etc.) se pueden eliminar con la orden **`strip`**. Esto reduce el tamaño del ejecutable. Podemos ver un ejemplo:

```
$ file hola; ls -l
hola: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.4, not stripped
-rwxr-xr-x 1 usuario1 users 10063 nov 29 10:37 hola
$ strip hola; file hola; ls -l hola
hola: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.4, stripped
-rwxr-xr-x 1 jagomez users 5980 nov 29 10:38 hola
```

5. Bibliotecas

Una biblioteca es una colección de objetos, normalmente relacionados entre sí. En el sistema existen un conjunto de bibliotecas predefinidas que dan servicios a las aplicaciones o a los servicios del sistema operativos. El usuario también pude definir sus propias bibliotecas al objeto de organizar mejor los módulos de sus aplicaciones y facilitar que éstas compartan módulos.

Como vimos cuando describíamos el ciclo de vida de un programa, el enlazador recibe, además de los archivos objeto, una serie de bibliotecas que utilizará para construir el ejecutable. Los módulos del programa pueden incluir referencias a símbolos definidos en algún objeto de una biblioteca ya sea una función o una variable exportada por la misma. Por tanto, estos son extraídos por el enlazador. Como resultado final, el ejecutable contiene todas las secciones del mismo tipo, con independencia de si provienen de un módulo de la aplicación o de una extraído de la biblioteca.

La Figura 15 muestra una clasificación de las bibliotecas en función de su implementación. En lo que resta de apartado, caracterizaremos cada uno de los tipos y comentaremos los elementos básicos de su construcción y uso.

Las bibliotecas estáticas están enlazadas con el programa y forman parte de él (Figura 16a). Tienen como extensión *.a.* Las bibliotecas de objetos dinámicamente enlazados (archivos .so) se enlazan en tiempo de ejecución y son compartidas por los programas que las utilizan (Figura 16b). Pueden ser:

- Enlazadas en tiempo de ejecución pero consciente de que son estáticas, es decir, deben estar

disponibles durante la fase de compilación/enlazado. Los objetos compartidos no están incluidos en el ejecutable pero son ligados en ejecución. Este tipo de bibliotecas son las DLL que se utilizan en Windows.

- Cargados dinámicamente y enlazados durante la ejecución (por ejemplo, plug-ins de un navegador) usando las funciones del cargador-enlazador dinámico del sistema. Son las bibliotecas que soportan los archivos ELF.

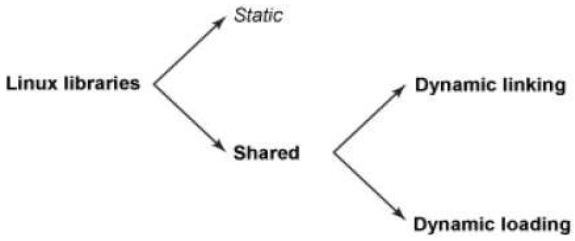


Figura 15.- Clasificación de las bibliotecas.

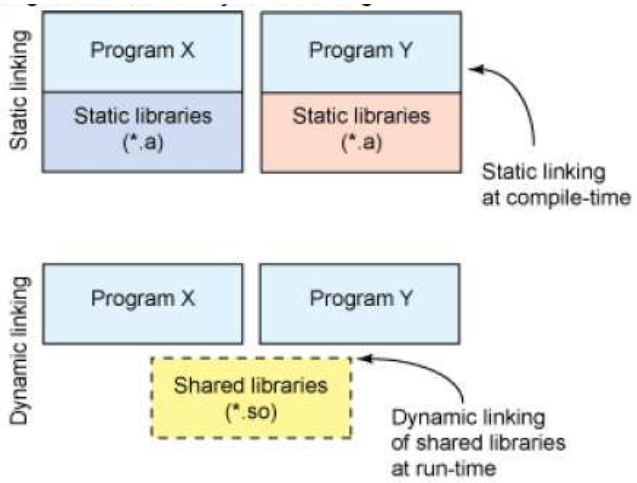


Figura 16.- Bibliotecas dinámicas frente a estáticas.

5.1 Bibliotecas estáticas

Para mejorar la modularidad y reusabilidad de los programas, las bibliotecas incluyen las funciones utilizadas comúnmente. Una bibliotecas estática es básicamente un conjunto de archivos objeto que se copia en un único archivo, que es la biblioteca estática.

Para mostrar los pasos básicos para la creación de una bibliotecas estática, vamos a partir de un sencillo ejemplo de una biblioteca con una función cuyo código es:

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

double media(double a, double b) {
  return (a+b) / 2;
}
```

El archivo `cal_media` contiene una función media que toma dos valores tipo `double` (utilizado para almacenar números reales) y devuelve su valor medio. Utilizamos como archivo de cabecera `cal_media.h` cuyo contenido es

```
double media(double, double);
```

Este archivo se crea con la orden **ar** (*archiver*), pero previamente debemos convertir el archivo de código en un archivo objeto:

```
$ gcc -c calc_mean.c -o calc_mean.o
```

Invocamos al archivador para producir la biblioteca estática, con nombre `libmean.a`[1], pasándole como argumento el archivo objeto generado en el paso anterior:

```
$ ar rcs libmean.a  calc_mean.o
```

La biblioteca puede generarse a partir de más de un archivo objeto en la forma

```
$ ar cr libfuncion.a funcion1.o funcion2.o
```

La biblioteca estática de funciones `libfuncion.a` no es más que una colección de archivos objetos encadenados juntos con una tabla de contenidos para una acceso rápido a los símbolos.

Para ver cómo se utilizan, vamos a definir un programa ejemplo que usa la función definida en la biblioteca. El código del programa es:

```
$ cat prueba.c
#include <stdio.h>
#include "calc_mean.h"

int main(int argc, char* argv[]) {

  double v1, v2, m;
  v1 = 5.2;
  v2 = 7.9;

  m  = media(v1, v2);

  printf("The mean of %3.2f and %3.2f is %3.2f\n", v1, v2, m);

  return 0;
}
```

Enlazamos frente a la biblioteca estática:

```
$ gcc -static prueba.c -L. -lmean -o statically_linked
```

Cuando, durante el proceso de enlazado incluimos una biblioteca estática, el enlazador da una pasada por la biblioteca y añade todo el código/datos correspondientes a los símbolos usados en el programa. El enlazador ignora los símbolos no referenciados de la biblioteca y aborta con error cuando encuentra un símbolo redefinido.

Un aspecto que se suele pasar por alto del enlazado es que muchos compiladores suministran un pragma para declarar ciertos símbolos como débiles. Por ejemplo, el siguiente código declara una función que el enlazador incluirá sólo si no está definida en cualquier otro sitio:

```
#pragma weak mi_funcion
/* Solo la incluye el enlazador sino no lo está en otro sitio */
void mi_funcion() {
. . .
}
```

Alternativamente, podemos incluir un pragma weak para forzar al enlazador a ignorar símbolos sin resolver. Por ejemplo, el siguiente programa

1 El nombre de la biblioteca debe comenzar con las letras *lib* y tener sifijo *.a*.

2 Jay Conrod, *Tutorial on Function Interposition in Linux*, 2009, disponible en

```
#pragma weak debug
extern void debug (void);
void (*debugfunc) (void) = debug;
int main() {
       printf("Hola Mundo\n");
       if (debugfunc) (*debugfun)();
;
```

Este código se compila y enlaza esté, o no, definida la función `debug()` en algún archivo objeto. Si el símbolo no esta definido, el enlazador normalmente lo sustituye con 0. Así, esta técnica es un método útil de invocar un código opcional que no requiere recompilar la aplicación completa (contrastar esto con habilitar la característica opcional a través de una macro del preprocesador).

Si bien las bibliotecas estáticas son fáciles de crear y manipular, presentan ciertos problemas de mantenimiento del software y de utilización de recursos. Por ejemplo, cuando el enlazador incluye una biblioteca estática en un programa, este copia los datos de la librería en el programa. Si es necesario modificar (parchear) la biblioteca, todos los programa que utilizan esta biblioteca debe ser recompilados de nuevo para que actualicen con la nueva versión de la biblioteca. También, copiar la biblioteca en cada programa aumente el espacio de memoria y disco consumido, especialmente para la bibliotecas frecuentemente usadas como las de C (Figura 16a).

5.2 Bibliotecas dinámicas

Del proceso explicado hasta el momento, el código del ejecutable es autocontenido, es decir, incluye todo lo necesario para su ejecución. Esto tiene varias desventajas: (a) el ejecutable puede ser bastante grande; (b) el programa que usa una biblioteca contiene el código de la misma y éste estará en todos los ejecutables que la usan, malgastando disco y memoria; y (c) para beneficiarnos de la actualización de una biblioteca deberemos recompilar los programa que la utilizan. Para resolver estas deficiencias se utilizan las bibliotecas montadas dinámicamente, o simplemente bibliotecas dinámicas.

Ya que las bibliotecas dinámicas se integran en ejecución, estás deben estar ya organizadas en regiones, es decir, ya se ha realizado en ellas la reubicación de módulos (a diferencias de las estáticas). Por tanto, las genera el montador al que se le suministra una opción para que genere una biblioteca y no un ejecutable. Las principales diferencias entre un ejecutable y una biblioteca dinámica son que ésta última contendrá información de reubicación y una tabla de símbolos (ya que en ejecución hay que realizar la resolución de símbolos y la reubicación de regiones), y su cabecera no tiene información de cuál es el punto de inicio ya que no tiene sentido.

- **Bibliotecas compartidas de carga dinámica**

Dentro de las bibliotecas dinámicas distinguiremos entre las bibliotecas compartidas de carga dinámica, donde la reubicación de símbolos y la reubicación se realiza en tiempo de enlazado, y las bibliotecas compartidas enlazadas dinámicamente, donde las operaciones de enlazado se realiza en tiempo de ejecución (Figura 16).

Este tipo de bibliotecas se encuentra principalmente con dos problemas de reubicación, a saber:

1. Las referencias incluidas en la biblioteca a su propios símbolos estáticos, código o datos, deben ajustarse para que estén acordes con las zonas del mapa de memoria del proceso donde se ha cargado la biblioteca, que puede ser diferente para cada proceso.
2. Las referencias del programa a los símbolos de la biblioteca, y las de la propia biblioteca a los símbolos de otras bibliotecas anteriormente cargadas y enlazadas, deben resolverse y ajustarse de acuerdo con su ubicación en el mapa del proceso.

Respecto al primer tipo reubicación de referencias, se trata de un problema de autorreferencias: si dos procesos comparten una biblioteca haciéndola corresponder con rangos diferentes de sus mapas de direcciones, no podrá haber una referencia dentro de la biblioteca a otra parte de la misma (en este caso una referencia a un símbolo estático de la propia biblioteca). Una posible solución es establecer un rango de direcciones predeterminadas y específico para cada biblioteca dinámica, de forma que todos los procesos que la utilizan la incluyen en ese rango. Un sistema que utilice esta solución debe tener un procedimiento de asignación de direcciones para cada biblioteca, y el enlazador lo usará en su creación. Obsérvese que el enlazador realiza directamente la etapa de reubicación de regiones sobre la biblioteca, por lo que no es necesaria en tiempo de ejecución.

Si bien la solución anterior elimina el problema de raíz al desaparecer la causa del mismo, es poco flexible, ya que limita el número de bibliotecas que pueden existir en un sistema y puede provocar que el mapa de memoria de un proceso sea muy grande y disperso (muchas zonas sin usar). Una solución más flexible, que se utiliza en Windows, consiste en cargar la biblioteca en una posición asignada pero, si se produce conflicto al existir otra en esa posición, se carga en otro lugar y se reajustan las direcciones. Se tiene que rehacer la reubicación de regiones, perdiendo la posibilidad de compartirla. Esta situación es sostenible por que la posibilidad de conflicto es baja.

Respecto a la segunda cuestión de reubicación, se procede como en las bibliotecas estáticas al resolver y reubicar en tiempo de enlace pero la biblioteca no se incluye en el ejecutable. Así, en ejecución no se realiza montaje, sino que simplemente se carga la biblioteca. El problema que plantea esta solución es el de las actualizaciones de la biblioteca.

Para manejar las actualizaciones podemos hacer que el enlazador cree automáticamente una tabla de las funciones y variables exportadas en cada biblioteca, de forma que las referencias a las mismas se resuelvan con un acceso indirecto a través de la tabla correspondiente. De esta forma, actualizar una función no afecta a la dirección de ningún símbolo exportado. Añadir una nueva función requiere incluir un una nueva entrada en la tabla, manteniendo el resto igual. Debemos tener presente que el objetivo de la indirección no es facilitar la reubicación en ejecución, como ocurre en las bibliotecas enlazadas dinámicamente que veremos en siguiente punto, puesto que la reubicación se ha completado en el enlazado. El propósito de la indirección es poder crear nuevas versiones de una biblioteca en la que los símbolos puedan están ubicados en diferentes posiciones. El programa que utiliza la biblioteca sigue funcionando en tanto en cuanto no se mantenga la posición de cada símbolo en la tabla.

Este acceso indirecto plantea un problema técnico: dado que el compilador no conoce si una determinada referencia externa corresponderá en el enlazado a un símbolo definido en otro módulo, en una biblioteca estática o una dinámica, no puede saber si tiene o no que generar un acceso indirecto para acceder a un símbolo, pues este sería superfluo en los dos primeros casos. Para resolverlo, distinguiremos si se trata de una función o una variable.

En el caso de una referencia a una función de la biblioteca, la solución es utilizar una *función tocón* (stub) que esconda la indirección. De esta forma, el enlazador resuelve la referencia como una llamada directa a la función tocón. Dentro de esta función, el enlazador incluye un salto indirecto usando la dirección almacenada en la posición correspondiente de la tabla de funciones exportada por la biblioteca. Como la función tocón no incluye una instrucción de retorno, la función real devuelve el control directamente al programa sin pasar por el resguardo.

Si se trata de una referencia a una variable, es más complicado. En general se requiere que el programador lo especifique explícitamente, como ocurre en Windows con el calificador `dllimport` para las variables externas de bibliotecas dinámicas.

La Figura 17 ilustra está técnica para mostrar cómo se resuelve el acceso por parte de un programa a dos símbolos de una biblioteca: una variable global y una función.

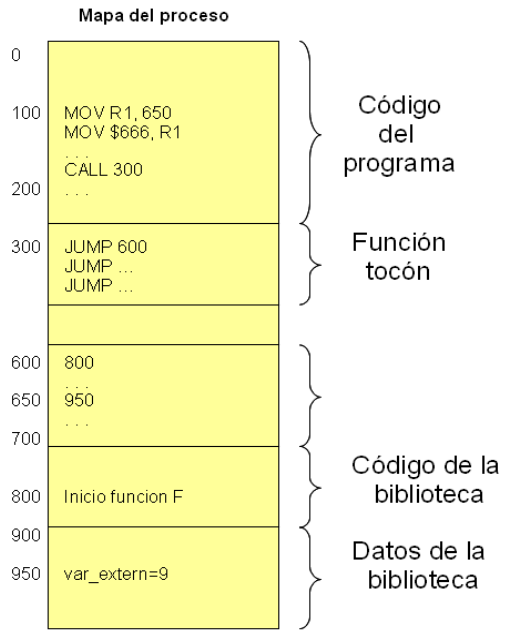


Figura 17.- Programa que usa una biblioteca compartida de carga dinámica.

- **Bibliotecas compartidas enlazadas dinámicamente**

Este tipo de bibliotecas ofrece una solución más flexible y con mayor funcionalidad que las anteriores. Los SO que utilizan el formato ELF se engloban dentro de esta categoría (de hecho es el principal objetivo de este formato). La principal flexibilidad radica en que permiten construir técnicas sofisticas como la **interposición**[2], que permite determinar en ejecución a que símbolo se está refiriendo un programa y, por tanto, la posibilidad de cambiarlo.

Para solventar el problema de las autorreferencias, debemos generar código PIC, utilizando un direccionamiento relativo al contador de programa. De esta forma la biblioteca no se ve afecta por la posición de memoria en la que se ejecuta.

En cuanto a la resolución de referencias, lo más frecuente es usar un direccionamiento indirecto a través de una tabla almacenada en la zona de datos, de forma que el proceso de resolución y reubicación actualice la tabla, pero no el código del programa o la biblioteca. En sistemas basado en ELF, esta tabla se denomina **Gobal Offset Table** (GOT). Durante el proceso de enlazado de un programa o biblioteca, se determina cuáles son los símbolos externos usados y se habilita una tabla del tamaño requerido en la región de datos. A cada uno de los símbolos se le hace corresponder con una posición en la tabla y las referencias a los mismos en el programa o biblioteca se convierten un acceso indirecto a través de la posición de la tabla asignada a ese símbolo. La resolución final de los símbolos se produce en ejecución. Por ello, por cada símbolo pendiente de resolver, se determina que posición le ha correspondido en el mapa en el proceso de carga y se actualiza la posición de la tabla correspondiente.

En este caso, el uso de la tabla de indirecciones tiene un propósito y características diferentes al de las bibliotecas que vimos en el punto anterior. Ahora, la indirección se resuelve en ejecución y el contenido de la tabla no es constante y se rellena en ejecución.

Como sucedía en el caso anterior, dado que al compilar un programa no se sabe si una referencia externa se corresponderá con un símbolo de una biblioteca dinámica, no es posible generar el acceso indirecto

2 Jay Conrod, *Tutorial on Function Interposition in Linux*, 2009, disponible en http://www.jayconrod.com/cgi/view_post.py?23.

requerido. Para resolver este problema tenemos que:

- Si la referencia es a una función, la solución es similar a las funciones tocón salvando la diferente naturaleza de la indirección. En el caso de referencias de una programa, pueden utilizarse referencias absolutas a la posición de la tabla. Ahora bien, si se trata de una referencia de una función de una biblioteca a una función definida en otra, la referencia a una posición de la tabla debe ser relativa (por ejemplo, respecto al contador de programa) para asegurar el comportamiento PIC. En el caso de formato ELF, esta tabla se denomina **Procedure Linkage Table** (PLT). Esta tabla se engloba, a diferencia de la de indirección, en la sección de código, ya que no puede modificarse en tiempo de ejecución. Volveremos a este tema en breve. La Figura 18 muestra como PLT añade un nivel de indirección para las llamadas a funciones de forma similar a lo que hace GOT para datos.
- En cuanto a las referencias directas a variables de la biblioteca desde el programa, lo habitual para evitar la indirección es asociar la variable externa a la región de datos del programa en lugar de a la biblioteca. Por tanto, en la tabla de indirecciones solo aparecen entradas relacionadas con llamadas a funciones definidas en la biblioteca. Respecto de las referencias a variable globales desde una biblioteca, en todas se usará un nivel de indirección.

En resumen, con esta técnica se sustituye una reubicación en la parte del código, que impedía compartir código del programa y de las bibliotecas, por una reubicación en la parte de datos, que no afecta a ese posible comportamiento. La Figura 18 muestra el uso de la biblioteca dinámica por parte de un programa.

Desde el punto de vista de la eficiencia, un aspecto importante de las bibliotecas dinámicas es que el proceso de enlazado se produce en cada invocación del programa. Para minimizar la sobrecarga de ejecución, las bibliotecas utilizan tanto ***tablas de indirección*** como ***ligadura perezosa de símbolos***. Esto es, la ubicación de los símbolos externos realmente hace referencia a las entradas de la tabla, que permanecen sin ligar hasta que la aplicación realmente los necesita. Esto reduce el tiempo de arranque debido a que la mayoría de las aplicaciones sólo usan un subconjunto pequeños de funciones de la biblioteca.

La tabla de enlazado de procedimientos (PLT) permite implementar la ligadura perezosa. En la Figura 19, observamos un programa creado por el enlazador estático que referencia dos funciones definidas en sendas bibliotecas. Para hacer que la ligadura perezosa de símbolos funcione en ejecución, el enlazador dinámico limpia todas las entradas de la PLT y las hace apuntar a una función especial de ligadura de símbolos dentro del cargador dinámico de la biblioteca. Lo bello de esta idea es que cuando una función se utiliza por vez primera, el enlazador dinámico toma el control del proceso y realiza todas las ligaduras necesarias. Después de localizar un símbolo, el enlazador simplemente sobre-escribe la entrada de la PLT de forma que llamadas posteriores a la función transfieran el control a ésta en lugar de llamar de nuevo al enlazador. La Figura 20, ilustra el proceso.

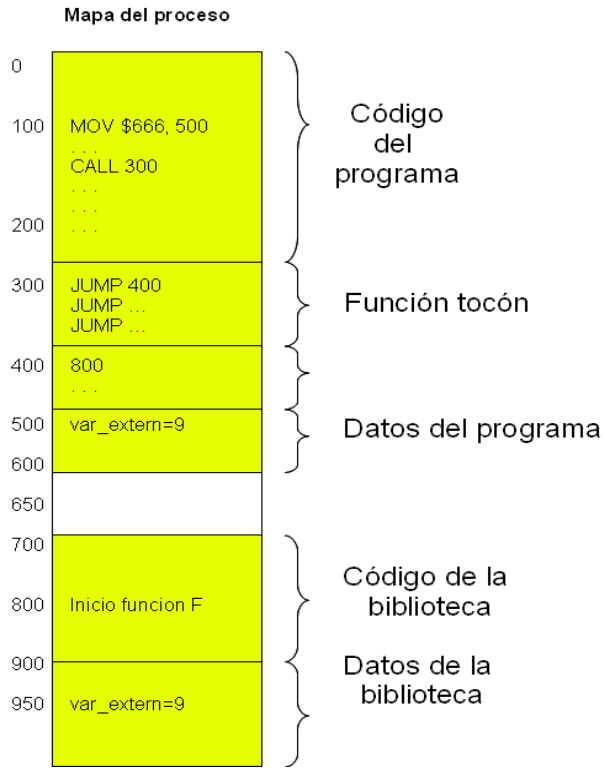


Figura 17.- Un programa usando una biblioteca enlazada dinámicamente.

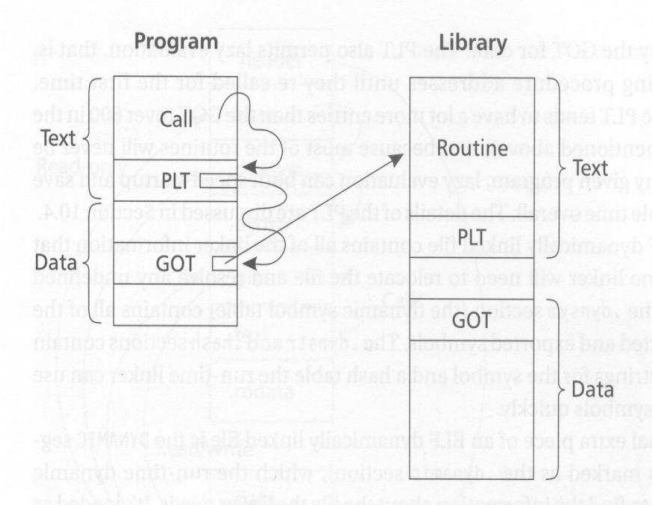


Figura 18.- PLT y GOT.

Aunque la ligadura de símbolos es en general transparente al usuario, podemos verla definiendo la variable de entorno LD_DEBUG al valor ligaduras antes de iniciar el programa. Un experimento interesante es ver las ligaduras dinámicas de aplicaciones interactivas tales como interpretes o navegadores, por ejemplo:

```
$ LD_DEBUG=bindings python
```

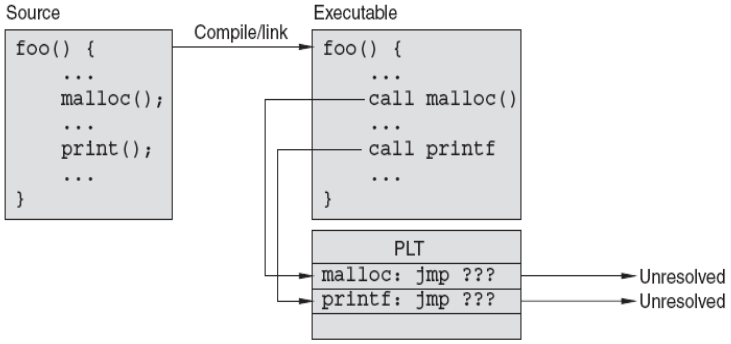
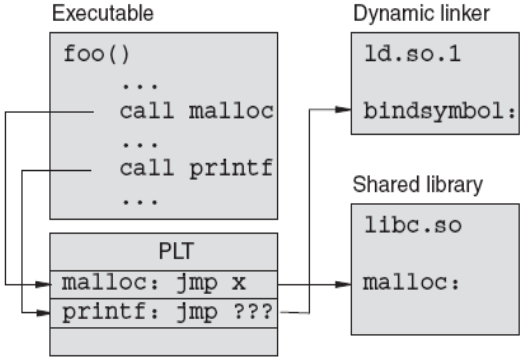


Figura 19.- Estructura interna de un ejecutable enlazado con una biblioteca compartida. Las llamadas a funciones de la biblioteca apuntan a las entradas de la PLT que están sin resolver.

Figura 20.- La ligadura dinámica de los símbolos de la biblioteca en bibliotecas compartidas: `malloc` es ligado a la biblioteca de C, y `printf` no se ha utilizado aún y está ligado al enlazador.



Retomando el ejemplo comentado para bibliotecas estáticas**,** ahora crearemos la biblioteca del ejemplo como compartida. Como ocurría con las bibliotecas estáticas, debemos crear el archivo objeto. Ahora, utilizamos la opción del compilador -fPIC que le indica que debe generar código independiente de la posición. Después, creamos la biblioteca:

```
$ gcc -c -fPIC calc_mean.c -o calc_mean.o

$ gcc -shared -Wl,-soname,libmean.so.1 -o libmean.so.1.0.1  calc_mean.o
```

Volviendo al ejemplo `prueba.c` que vimos en el apartado de bibliotecas estáticas, ahora podemos compilar el archivo `prueba.c` contra la biblioteca compartida recién creada con[3]:

```
$ gcc main.c -o dynamically_linked -L. -lmean
```

El enlazado en tiempo de ejecución permite un mantenimiento fácil de las bibliotecas. Ahora, si detectamos un error en una biblioteca, solo debemos modificarla y no es necesario recompilar los programas que la utilizan, estos utilizan la nueva biblioteca la próxima vez que se ejecuten. Un aspecto sutil de este tipo de bibliotecas es que permiten al SO hacer un cierto número de optimizaciones de memoria. Dado que las bibliotecas son básicamente código ejecutable que es no auto-modificable, el SO puede ubicarlo en regiones de memoria de solo lectura compartidas por todos los procesos utilizando técnicas de compartición de memoria como las vistas en paginación. Así, varios programas comparten una única copia de la biblioteca en memoria, con lo que se reduce su uso y mejora el rendimiento del sistema.

En la mayoría de los sistemas, el enlazador maneja ambos tipos de bibliotecas. Por ejemplo, consideremos un programa enlazado con diferentes bibliotecas:

```
$ gcc hola.c -lpthread -lm
```

Si las bibliotecas `lpthread` y `lm` han sido compiladas como bibliotecas compartidas, el enlazador comprueba los símbolos sin resolver e informa de los errores de la forma normal. Sin embargo, en lugar de copiar los contenidos de la biblioteca en el ejecutable, el enlazador simplemente registra el nombre de la biblioteca en una lista en el ejecutable. Podemos ver los contenidos de las dependencias de bibliotecas con la orden **`ldd`**:

```
$ ldd hola
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7552000)
        /lib/ld-linux.so.2 (0xb76f0000)
```

Cuando ligamos un símbolo en ejecución, el enlazador dinámico busca la biblioteca en el mismo orden en el que aparece en la linea de enlace y utiliza la primera definición encontrada del símbolo. Si más de una biblioteca define el mismo símbolo, solo se aplica la primera definición. La duplicidad de símbolo no suele ocurrir pues el enlazador estático escanea todas la bibliotecas e informa del error en éste caso. Si embargo, si se pueden dar si tenemos definiciones de símbolos débiles, si una modificación de una biblioteca existente añade un nuevo nombre que entra en conflicto con los existentes, o si la definición de la variable `LD_LIBRARY_PATH` invierte el camino de carga.

Por defecto, muchos sistema exportan todos los símbolos globalmente definidos en una biblioteca (cualquier cosa accesible mediante la declaración `extern` de C/C++). Sin embargo, ciertas plataformas, la exportación

3 Observar que ahora no es necesario especificar ni el prefijo *lib* ni el sufijo *.so*.

está controlada por una lista dada como una opción del enlazador o una extensión del compilador. Cuando se necesitan estas extensiones, el enlazador dinámico solo liga los símbolos que son explícitamente exportados.

Cargando las bibliotecas

Cuando se ejecuta un programa con bibliotecas compartidas, su ejecución no se inicia con la primera instrucción del mismo. En su lugar, el SO carga y ejecuta el enlazador dinámico (normalmente denominado ld.do), que escanea las listas de nombres de bibliotecas embebidas en el ejecutable. Estos nombres no se codifican nunca con pathnames absolutos. En su lugar, se usan nombres simples tales como `libpthread.so.0`, *`libm.so.6`*, etc. donde el dígito final representa la versión. Para localizar la bibliotecas, en enlazador dinámico utiliza un path de búsqueda configurable. El valor por defecto de este path se almacena normalmente en un fichero de configuración de sistema, como `/etc/ld.so.conf`. Además, otros directorios de búsqueda pueden embeberse en el ejecutable o a través de la variable de entorno LD_LIBRARY_PATH.

Las bibliotecas dinámicas siempre se cargan en el orden en el que fueron enlazadas. Por tanto, si enlazamos nuestro programa con

```
$ cc $SRCS -lfoo -lbar -lsocket -ls -lc
```

El enlazador dinámico carga las bibliotecas en el orden *libfoo.so*, *libbar.so*, *libsocket.so*, *libm.so*, etc. Si las bibliotecas compartidas incluyen dependencias adicionales de bibliotecas, estas bibliotecas se añaden al final de la lista durante la carga. Para ser más preciso, el orden de carga esta determinado para el primer recorrido en anchura de las dependencias de la biblioteca comenzando por las bibliotecas directamente enlazadas con el programa. Internamente, el cargador mantiene la pista de estas bibliotecas poniéndolas en una lista enlazada denominada *cadena de enlazado*. Es más, el cargador garantiza que una biblioteca solo se enlaza una vez.

Dado que recorrer los directorios es relativamente lento, el cargador no mira en los directorios de `/etc/ld.so.conf`, sino que en su lugar consulta un archivo caché, generalmente `/etc/ld-so.cache`, que es una tabla que iguala nombres de bibliotecas con pathnames. Si añadimos una biblioteca a `/etc/ld.so.conf` debemos reconstruir el archivo caché con la orden **`ldconfig`**, o el cargador no la encontrará.

Si el cargador dinámico no encuentra una biblioteca en la caché, realiza un último esfuerzo con una búsqueda manual entre los directorios de bibliotecas del sistema tales como `/lib`, `/usr/lib` antes de devolver un error. Esta conducta dependerá del sistema operativo.

Podemos obtener información detallada sobre como el cargador dinámico carga las bibliotecas ajustando la variable `LD_DEBUG` a `libs`, por ejemplo:

```
$ LD_DEBUG=libs a.out
```

Como indicabamos antes, podemos embeber el path de búsqueda de una biblioteca en el ejecutable con las opciones -R, -Wl, o -rpath del enlazador. Por ejemplo

```
$ cc $SCRS -W1,-rpath=/home/usuario/libs -L/home/usuario/libs -lfoo
```

En este caso, el programa encontrará *libfoo.so* en directorio correcto sin tener que definir ninguna variable de entorno especial.

En la discusión anterior, hemos excluido la posibilidad de carga de una biblioteca de forma explícita a través de los servicios del sistema operativo. En sistemas Unix, esta funcionalidad viene dada por las funciones `dlopen()`, `dlsym()`, y `dlclose()`.

6. Ejemplo del ciclo de vida de un programa en GNU/Linux

¿Por qué no vemos las tres primeras fases cuando compilamos? Para estudiar el proceso vamos primero a proponer un ejemplo sencillo que es una variante del programa "Hola Mundo", como se muestra a continuación

```
$ cat hola.c
#include "hola.h"
main()
{
       printf("hola … %d\n", VAR);
}
$ cat hola.h
#define VAR 100
```

Nosotros compilamos en general nuestro programa con gcc de la forma:

```
$ gcc -o hola hola.c
```

En realidad, `gcc` es un *wrapper* (envoltorio o programa que controla el acceso a otros programas) que invoca de forma ordenada a los programas que realizan cada una de las fases antes citadas. Podemos ver lo que hacer la orden con la opción -v:

```
$ gcc –v hola.c
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.6/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Debian 4.6.2-9' --with-
bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-
languages=c,c++,fortran,objc,obj-c++,go --prefix=/usr --program-suffix=-4.6 --enable-
shared --enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib --without-
included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.6 --
libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-
libstdcxx-time=yes --enable-plugin --enable-objc-gc --with-arch-32=i586 --with-
tune=generic --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu -
-target=x86_64-linux-gnu
Thread model: posix
gcc version 4.6.2 (Debian 4.6.2-9)
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/4.6/cc1 -quiet -v -imultilib . -imultiarch x86_64-linux-
gnu hola.c -quiet -dumpbase hola.c -mtune=generic -march=x86-64 -auxbase hola -version -o
/tmp/ccXXtkYR.s
GNU C (Debian 4.6.2-9) version 4.6.2 (x86_64-linux-gnu)
        compiled by GNU C version 4.6.2, GMP version 5.0.2, MPFR version 3.1.0-p3, MPC
version 0.9
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../../x86_64-
linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/4.6/include
 /usr/local/include
 /usr/lib/gcc/x86_64-linux-gnu/4.6/include-fixed
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
GNU C (Debian 4.6.2-9) version 4.6.2 (x86_64-linux-gnu)
        compiled by GNU C version 4.6.2, GMP version 5.0.2, MPFR version 3.1.0-p3, MPC
version 0.9
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: 37a487dbd7e6cd72b6da9f7b4bf1285d
```

```
hola.c: In function 'main':
hola.c:5:4: warning: incompatible implicit declaration of built-in function 'printf'
[enabled by default]
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 as --64 -o /tmp/ccnSWZNK.o /tmp/ccXXtkYR.s
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/4.6/:/usr/lib/gcc/x86_64-linux-
gnu/4.6/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/4.6/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/4.6/:/usr/lib/gcc/x86_64-linux-
gnu/4.6/../../../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/4.6/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/x86_64-linux-
gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/4.6/collect2 --build-id --no-add-needed --eh-frame-hdr -m
elf_x86_64       --hash-style=both       -dynamic-linker       /lib64/ld-linux-x86-64.so.2
/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../x86_64-linux-gnu/crt1.o   /usr/lib/gcc/x86_64-
linux-gnu/4.6/../../../x86_64-linux-gnu/crti.o                  /usr/lib/gcc/x86_64-linux-
gnu/4.6/crtbegin.o    -L/usr/lib/gcc/x86_64-linux-gnu/4.6    -L/usr/lib/gcc/x86_64-linux-
gnu/4.6/../../../x86_64-linux-gnu  -L/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../../lib  -
L/lib/x86_64-linux-gnu  -L/lib/../lib  -L/usr/lib/x86_64-linux-gnu  -L/usr/lib/../lib  -
L/usr/lib/gcc/x86_64-linux-gnu/4.6/../../.. /tmp/ccnSWZNK.o -lgcc --as-needed -lgcc_s --
no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-
gnu/4.6/crtend.o /usr/lib/gcc/x86_64-linux-gnu/4.6/../../../x86_64-linux-gnu/crtn.o
```

En negrita, hemos marcado la invocación a los programas que realiza y que son:

- **cpp1**: el preprocesador
- **cc**: el compilador
- **as**: el ensamblador
- **collect2:** que a su vez es otro wrapper la invocar al enlazador, **ld**.

La salida también muestra las numerosas opciones y argumentos con las que se invoca a cada una de estas órdenes. Por qué hacerlo así, en lugar de llamar separadamente a cada programa. Una razón es la conveniencia, la otra portabilidad. La herramienta `gcc` nos suministra un gran número de conmutadores portables que afectan al comportamiento de los programas compilados. Esto nos asegura que nuestros programas compila correctamente en varias plataformas.

A continuación, vamos a compilar nuestro programa paso a paso, para ver con más detalle cada una de las fases. Cuando pasamos un archivo a `gcc`, este pasa por cuatro etapas. En la primera de ellas, el preprocesador examina los archivos `.c` en busca de declaraciones que comiencen por `#`. Cuando son de tipo `#include` archivo, localiza el archivo y lo añade al programa. Después, el preprocesador busca y sustituye todas las macros (`#define`). Una vez realizadas todas las sustituciones se invoca al compilador. Ojo: el compilador no ve el archivo `.c` original si no la suma del `.c` y `.h` con la macros sustituidas. Esto puede producir algunos problemas serios si el archivo de cabecera contiene errores. Una macro que se comportan como función con errores puede generar bastante confusión: el compilador no sabe nada sobre cabeceras, por tanto, cuando ve un error informa del número de línea que él ve; así cuando comprobamos el `.c` original, podemos ver una línea completamente inocente.

```
$ gcc -save-temps hola.c
$ ls -l
total 36
-rwxr-xr-x 1 jrevelle jrevelle 6618 ene  7 20:35 a.out
-rwxr-xr-x 1 jrevelle jrevelle 6618 ene  7 20:26 hola
-rw-r--r-- 1 jrevelle jrevelle   64 ene  7 20:26 hola.c
-rw-r--r-- 1 jrevelle jrevelle   16 ene  7 20:25 hola.h
-rw-r--r-- 1 jrevelle jrevelle  140 ene  7 20:35 hola.i
-rw-r--r-- 1 jrevelle jrevelle 1488 ene  7 20:35 hola.o
-rw-r--r-- 1 jrevelle jrevelle  451 ene  7 20:35 hola.s
El archivo generado por el preprocesador:
$ cat hola.i
# 1 "hola.c"
# 1 "<built-in>"
```

```
# 1 "<command-line>"
# 1 "hola.c"
# 1 "hola.h" 1
# 2 "hola.c" 2

main()
{
   printf ("hola ... %d\n", 100);
}
```

Podemos ver el código en ensamblador (el ensamblador se puede generar solo con `gcc -S)`:

```
$ cat hola.s
        .file   "hola.c"
        .section        .rodata
.LC0:
        .string "hola ... %d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $100, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 4.6.2-9) 4.6.2"
        .section        .note.GNU-stack,"",@progbits
```

Pasemos a examinar el enlazador, para ello utilizamos la opción `-c` indica que no se realice el enlazado, es decir, que solo genere el archivo objeto `hola.o`:

```
$ gcc –c hola.c
```

El paso final, es enlazar el archivo:

```
$ ld hola.o –o hola
ld: warning: cannot find entry symbol _start; defaulting to 08048074
hola.o: In function `main':
hola.o(.text+0x11): undefined reference to `printf'
```

Pero, de ¿dónde viene el error? Hemos utilizado la función `printf` de la biblioteca `libc.so` del directorio `/usr/lib`. Por tanto, debemos instruir al enlazador para que localice el código:

```
$ ld hola.o –o hola /usr/lib/libc.so
```

ó bien

```
$ ld hola.o –o hola –lc
ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

Dejando a un lado el aviso, si intentamos ejecutar nuestro programa, el shell se quejará indicándonos de que no encuentra el archivo (aunque este existe). El problema no es el shell. El shell es una interfaz con el sistema y no sabe como cargarlo ni ejecutarlo. Necesita los servicios de cargador de archivos ELF, `ld-linux.so.2`. Al enlazador le debemos que cargador añadir al ejecutable cuando se enlaza. Por tanto, enlazaremos nuestro programa:

```
$ ld hola.o –o hola –lc ld hola.o –o hola –lc-dymamic-linker /lib/ls-linux.so.2
ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

Cuando ejecutamos nuestro programa, `main` no es el primer fragmento de código que se llama. Se ejecuta primero el *código de arranque* que es quien invoca al `main`. Por tanto, debemos indicar a `ld` donde está el `main`. Realmente, la primera línea ejecutable dentro de un programa esta etiquetada como `_start`, por convenio. Debemos indicar, por consiguiente, a `ld` que utilice `main` como el inicio de nuestro programa.

```
$ ld –o hola hola.o –lc –e main –dynamic-linker /lib/ld-linux.so.2
$ ./hola
Hola … 100
Segmentation fault (core dumped)
```

¡Bueno, persisten los problemas! Hasta ahora nos hemos ocupado del arranque del programa, consideremos ahora su finalización. Para que nuestro programa termine correctamente, debemos incluir la llamada al sistema *exit()* que termina la ejecución de un programa (cuando se compila directamente, no paso a paso, el enlazador incluye esta función al detectar la última instrucción del programa o el return final).

```
$ cat hola.c
#include "hola.h"
main()
{
       printf("hola … %d\n", VAR);
       exit(1);
}
$ gcc –c a.c
$ ld –o hola hola.o –lc –e main –dymanic-linker /lib/ld-linux.so.2
$ ./hola; echo $?
Hola… 100
1
```

Cuando se lanza la ejecución de un programa (llamada al sistema `exec()` en Unix, o `CreateProcess()` en Windows) se crea el mapa de memoria de este en el que se crean las regiones de memoria de acorde a la información contenida en el ejecutable.

El cargador debe:

1. Validar los acceso y memoria – el cargador que es parte del SO lee la cabecera del ejecutable y realiza la validación de tipos, permisos de acceso y requisitos de memoria.
2. Construye el procesos – esto incluye:
   - Asignar memoria principal para la ejecución del programa.
   - Copiar el espacio de direcciones desde disco a memoria principal.
   - Copiar las secciones `.text` y `.data` del archivo ejecutable en memoria.
   - Copiar los argumentos de invocación del programa en la pila.
   - Inicializar los registros: ajusta SP para que apunte a la pila, y limpia el resto.
   - Salta a la dirección de inicio, que copia los argumentos salvado en la pila, y salta al `main()`.

Para comprender los detalles que se esconden tras el procedimiento de carga realizado por `execve`, echemos un vistazo la ejecutable ELF:

```
$ readelf –l hola
Elf file type is EXEC (Executable file)
Entry point 0x8048360
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004f5 0x004f5 R E 0x1000
  LOAD           0x0004f8 0x080494f8 0x080494f8 0x000e8 0x00100 RW  0x1000
  DYNAMIC        0x000540 0x08049540 0x08049540 0x000a0 0x000a0 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.got
.rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .ctors .dtors .got .dynamic .bss
   04     .dynamic
   05     .note.ABI-tag
```

La salida muestra la estructura general del proceso `hola`. La primera cabecera de programa corresponde al segmento de código del proceso, que se cargará del archivo desde el desplazamiento 0x000000 en una región de memoria que será proyectada en el espacio de direcciones del proceso en la dirección `0x08048000`. El segmento de código tendrá `0x004f5` bytes y debe estar alineado a página (`0x1000`). Este segmento comprenderá los segmentos ELF `.text` y `.rodata`, discutidos anteriormente, más segmentos adicionales generados durante el procedimiento de enlazado. Como esperábamos, está marcado como sólo lectura (R) y ejecutable (E), pero no se puede escribir (W).

La segunda cabecera de programa corresponde al segmento de datos. La carga de este segmento sigue los mismos pasos mencionados anteriormente. Sin embargo, observar que la longitud del segmento es de tamaño `0x000e8` sobre el archivo y `0x00100` en memoria. Esto se debe a la sección `.bss`, que se rellena a ceros y por tanto no debe estar presente en el archivo. Este segmento también debe estar alineado a página y contiene los segmentos ELF `.data` y `.bss`. Esta marcado como lectura/escritura (RW). La tercera cabecera de programa resulta del procedimiento de enlazado y es irrelevante a nuestra discusión.

Si disponemos de un sistema de archivos `/proc` podemos comprobar esto mientras se ejecuta el proceso "Hola Mundo":

```
$ cat /proc/`ps –C hola – pid h`|tr " " "\null"/maps
08048000-08049000 r-xp 00000000 08:05 277304     /root/hola
08049000-0804a000 rw-p 00000000 08:05 277304     /root/hola
40000000-40016000 r-xp 00000000 08:05 271030     /lib/ld-2.2.2.so
40016000-40017000 rw-p 00015000 08:05 271030     /lib/ld-2.2.2.so
40017000-40018000 rw-p 00000000 00:00 0
40018000-40019000 rw-p 00000000 00:00 0
40026000-4014c000 r-xp 00000000 08:05 350657     /lib/i686/libc-2.2.2.so
4014c000-40152000 rw-p 00125000 08:05 350657     /lib/i686/libc-2.2.2.so
40152000-40156000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

La primera región proyectada es el segmento de código del proceso, el segundo constituye el segmento de datos (datos + bss + heap), y el último, que no tiene correspondencia en el archivo ELF, es la pila. Se puede obtener información adicional mediante `time`, `ps`, y `/proc/pid/stat` de GNU.

7. Automatización del Proceso de Compilación y Enlazado. Herramientas y Entornos

Automatizar la construcción es la técnica utilizada durante el ciclo de vida de desarrollo software donde el código fuente de la aplicación es transformado en lenguaje máquina con un guión (script) para la construcción. Este proceso de automatización de la construcción es una práctica común conforme crece la complejidad del software.

Hay varias etapas en el ciclo de vida del desarrollo software para aplicaciones complejas. Primero, el código de la aplicación se desarrolla, se prueba, y se integra en un entorno de desarrollo autónomo. Una vez completado el software por parte del desarrollador, este se integra en un entorno compartido, donde deber ser integrado con otros componentes software que pueden haber sido construidos por otros desarrolladores. Antes de integrar el software en el entorno compartido, hay que tomar varias precauciones para asegurarse que el nuevo código no afecta al código desarrollado por otros miembros del equipo.

Típicamente la automatización de la construcción se completa utilizando un lenguaje de guiones que permite al desarrollador enlazar otros módulos y procesos dentro del proceso de compilación. Este lenguaje de guiones encapsula las tareas manuales para generar el producto software final. Estas tareas incluyen la documentación, prueba compilación, y distribución del código binario.

Nosotros veremos en las sesiones de prácticas parte de estas tareas, en concreto, la de compilación del código fuente en el correspondiente ejecutable. En concreto utilizaremos la herramienta **`make`**, y en especial, aprenderemos a construir los `makefiles`, que son los guiones que establece que acciones debe realizar la herramienta.

La automatización de la construcción es una bien conocida y buena práctica para generar software de calidad. La automatización de la prueba de código es la principal razón para esta mejora. Forzando la ejecución de guiones de prueba antes de realizar la integración de los módulos de código dentro del código compilado, el proyecto de software probablemente tendrá pocos errores durante la fase de uso.

Conforme la automatización ha avanzado, también han avanzado los lenguajes de guiones utilizados por los desarrolladores para compilar código fuente. Hoy en día, estos lenguajes de guiones realmente se han embebido dentro de los Entorno de Desarrollo Integrado (IDE), como el usado en la asignatura de Programación.

Otro beneficio clave de la automatización de la construcción es la habilidad de seguir la pista del impacto del código basado en compilaciones históricas. Una tarea dentro del guión de construcción incluye la generación de número de versión para el código. Este número de versión permite al desarrollador y testeador del software tener un punto de referencia sobre cuando se ha introducido un error en el entorno de producción.