Unidad 4 – Segunda parte

El lenguaje de ensamblador es una representación simbólica del lenguaje de máquina para un procesador en específico.

El ensamblador realiza dos pasadas mediante el proceso de traducción. Primero transforma el código fuente (código assembler) al código destino (código de máquina). Esto sucede para cualquier lenguaje de programación compilable. El producido del código ensamblado / compilado es el archivo código objeto.

Las pasadas significa abrir el código fuente y lo lee linea por linea para generar algo. En la segunda pasada vuelve a hacer el barrido y utiliza el resultado de la primer pasada para completar la traducción.

En la primer pasada se genera una tabla de símbolos (listing de ensamblado). Este es un archivo de texto que el ensamblador genera como resultante de la traducción. Una tabla de simbolos tiene filas y en cada una de ellas el ensamblador va poniendo las etiquetas del programa. El ensamblador genera esto en la primer pasada para generar la traducción. Cuando encuentra un simbolo en la segunda pasada, se asocia una dirección de memoria referente a donde se encontraba este simbolo. Cada etiqueta señala a una dirección en memoria particular y a partir de esta se genera una dirección relativa.

La tabla de simbolos se genera a partir del nombre del simbolo y la dirección relativa. Se genera a partir de esto un puntero “location counter”. Este location counter empieza en 0 con el 1er byte del código objeto ensamblado. Este contador se va actualizando, guardando solamente la longitud de cada instrucción de maquina, salteando las etiquetas y guardando solamente la información relevante.

Sabemos que la instrucción de máquina tiene una longitud distinta según en que procesador estemos. La instrucción en intel tiene un tamaño variable. En ARM tenemos un formato fijo, donde cada instrucción tiene el tamaño de 32 bits.

Cuando el ensamblador de intel lee la instrucción de máquina tiene que interpretar el opcode, el modo de direccionamiento y los operandos. Todo este trabajo lo hace para saber cuanto mide la longitud de la instrucción de máquina. De esta forma sabe cuantos bytes debe actualizar el Location Counter. Si la primera instrucción arranca de 0 y tiene 2 bytes de tamaño, la segunda instrucción tendrá comienzo a partir de los 2 bytes de acuerdo a su posición relativa. De esta manera para cada simbolo se le asigna un valor del location counter para que en la segunda pasada se tenga esta información a mano.

Segunda pasada:

En el caso de Intel se tiene que descifrar el opcode, modo de direccionamiento y operandos mientras que en ARM no, tal que ya por defecto se sabe que la instrucción mide 32 bits. Entonces para cada etiqueta se le otorga 4 bytes. Si la primera instrucción tiene una dirección relativa de 0, la segunda tendrá de 4 y así. La excepción a esta regla es cuando se definen areas de memoria específicas, donde el desplazamiento obviamente tendrá que ser calculado en base a esto.

En la segunda pasada en definitiva se termina de definir el código máquina necesario tras haber definido donde empieza y donde termina todo. La idea en la primera es dejar todo referenciado con el location counter que eso en definitiva es lo que le interesa al ensamblador en la segunda pasada para transformar a código máquina. Se realiza lo que sería la traducción. Cuando se realiza dicha traducción se genera el código objeto donde esta el código máquina.

La traducción primeramente transforma el mnemónico de la instrucción, en el opcode binario correspondiente. En algunos casos solo con el opcode se puede deducir el formato de la instrucción y posición y tamaño de cada uno de sus campos. Los compiladores son aquellos que tienen dicha información para realizar las traducciones.

Una vez que identifica el formato es capas de traducir cada nombre de operando en el registro o código de memoria apropiado. Todo valor en definitiva se convierte a código máquina.

Si en la traducción tengo una etiqueta/referencia en memoria lo que hago es remplazar dicha etiqueta por su equivalente escrito en la tabla de simbolos. En esta traducción también puedo haber indicadores de modo de direccionamiento, bits de codigo de condición, etc.

Código objeto

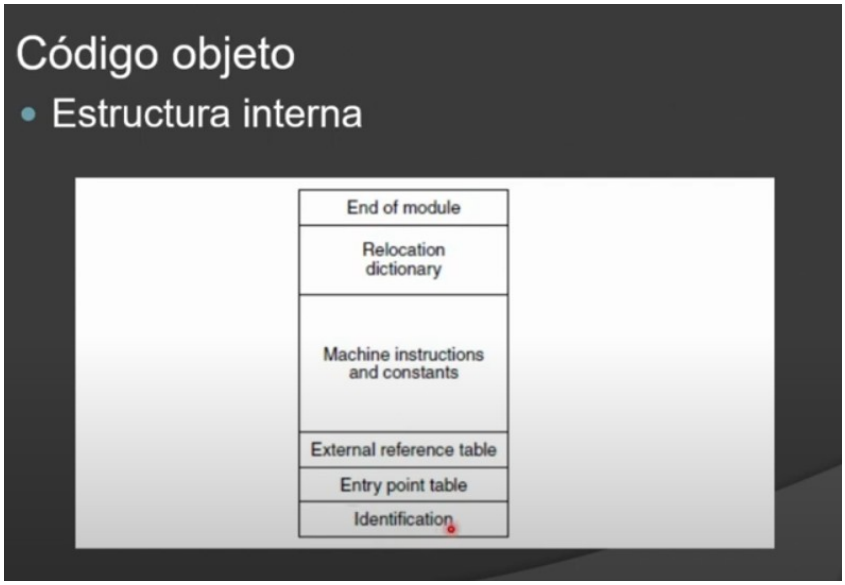
El código objeto es una representación del lenguaje de maquina del código fuente, generado por un compilador. El código ejecutable se crea despues por un linkeditor, algo aparte de lo que es el código máquina este. Cuando vos compilas/ensamblas no tenés todavia el ejecutable, internamente trabaja el link editor.

El código objeto contiene distintas secciones. La primera es una identificación (header). Contiene el nombre del módulo y sus respectivas longitudes. Luego se guarda la lista de símbolos que pueden ser referenciados desde otros módulos. Esto quiere decir que la tabla de punto de entrada no necesariamente tiene solamente lo que yo codeé, si no que guardo referencias a modulos (como funciones de C) para luego cuando se haga una link edición se pueda acceder a los mismos.

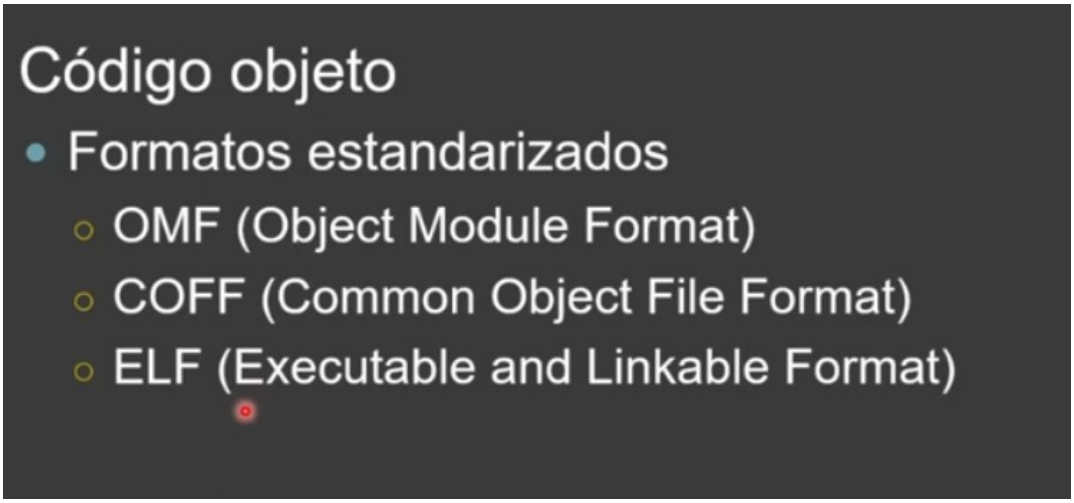
Después hay otra tabla de referencias externas, cuya lista de símbolos son los que se utilizan dentro del módulo pero se definen fuera de el (por ejemplo, sscanf). Mediante esta referencia a etiquetas externas luego puedo traerlas a partir de la link edicion.

El centro del código objeto es el código de máquina y las constantes (todo lo que definí en lenguaje ensamblador).

Por último se tiene un **diccionario de reubicabilidad**, que tiene una lista de referencias de campos en memoria que van a hacer necesarias que alguien más las traduzca en una dirección real efectiva. Esto sirve para transformar las direcciones relativas en direcciones reales.

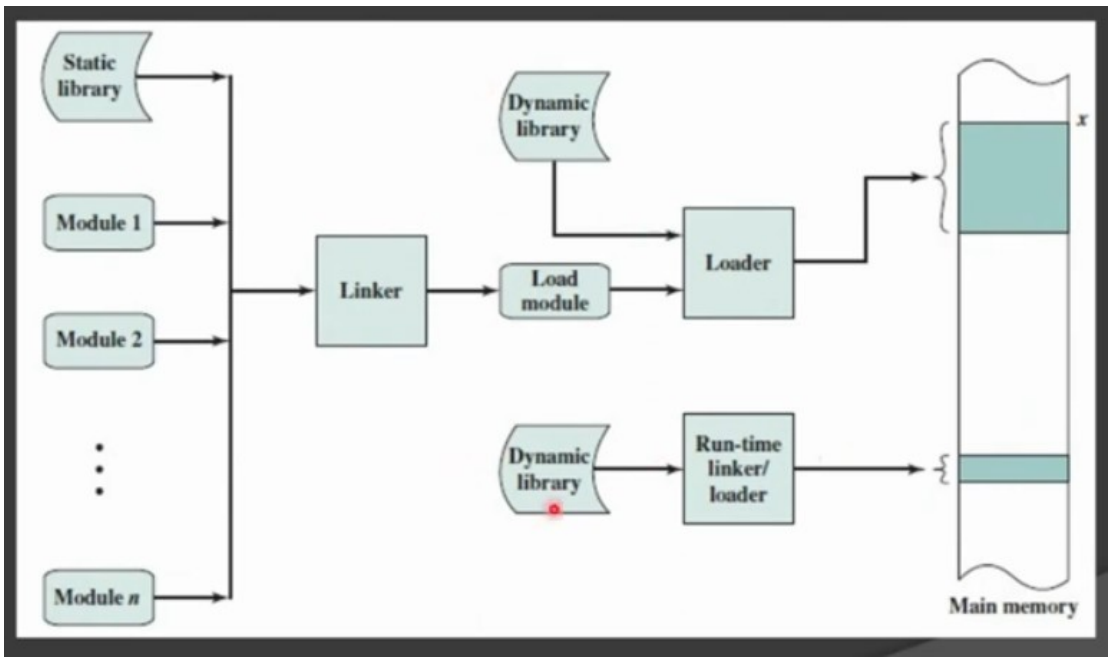


Los códigos objetos están estandarizados bajo un formato específico:



Link editor

La forma en la que el link editor trabaja esta resumido de esta forma. A la izquierda tengo cada código objeto (modules). Por lo general son varios. El linker agarra cada uno y carga cada modulo, generando el ".exe". Después a partir de este ejecutable (Load module) tengo opciones como el ensamblado dinámico. El loader tiene una rutina del sistema operativo, que es lo que hace que el sistema operativo transforme el código ejecutable como un proceso dentro de la memoria ram.

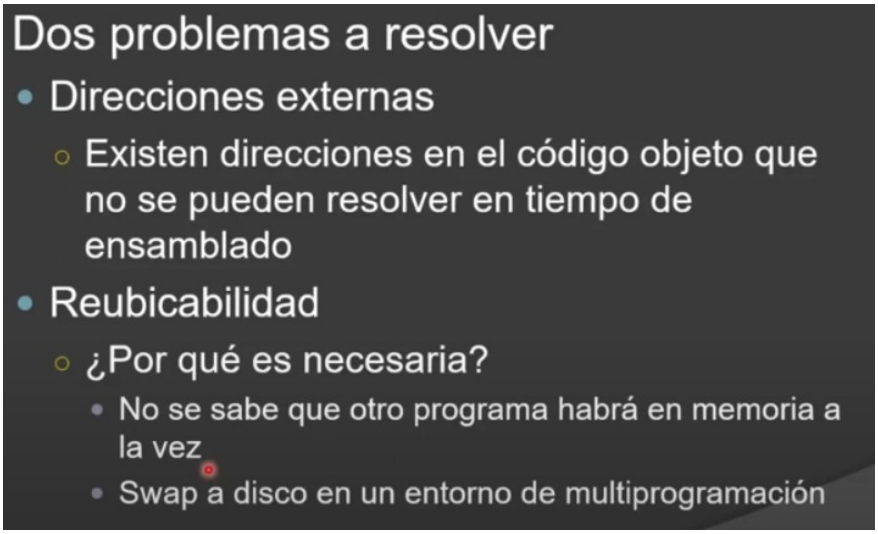


El linker es un programa, un software. Un programa que combina todos los código objeto y genera un único archivo ejecutable. El loader no es un programa, si no que es una rutina del sistema operativo que copia el ejecutable a la memoria principal, convirtiendolo en un proceso.

¿Por que es necesario todos estos pasos?. Lo que da sentido a separar el ensamblado / compilado de la generación del programa ejecutable es la existencia de las direcciones externas y la reubicabilidad del código.

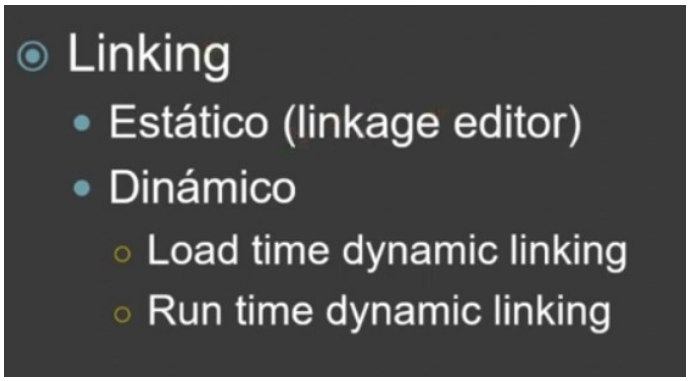
Las direcciones externas referencían a todo aquello que yo no programe y con lo que aún así interactuo. Todo eso es código pre-existente. Todo esto lo puedo usar por una invocación a partir de dicha dirección.

La posición relativa dentro del código es necesaria ya que dependiendo del sistema operativo lo que yo vaya a buscar puede estar ubicado en otro lado.



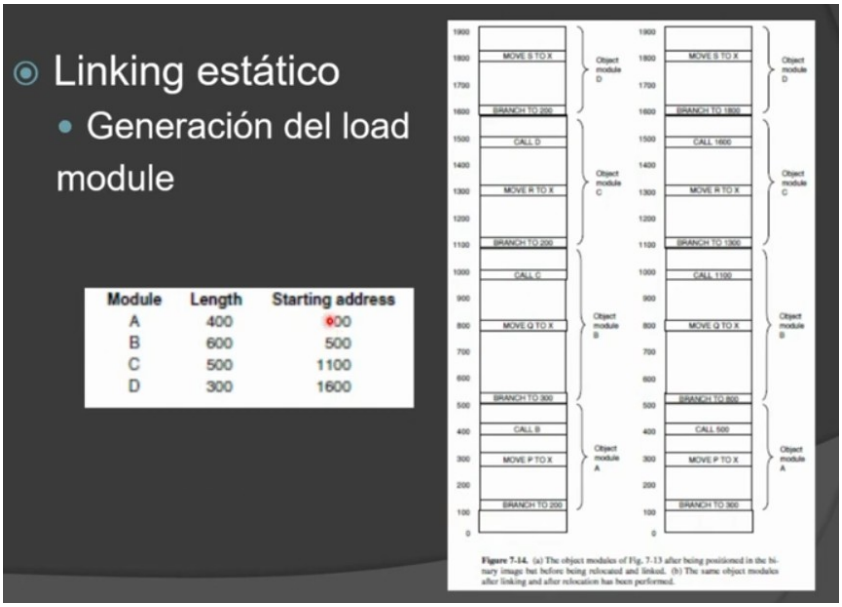
Linking

El linking tiene como entrada los códigos objetos y como salida el ejecutable. En principio exísten dos modos:



En el linkeo estático, cada módulo objeto que se crea tiene una referencia relativa al inicio del módulo. El link editor combina todos los modulos objetos y genera el código ejecutable que tiene todas las direcciones relativas, pero ya no en relación a cada modulo por separado, si no que ya se hace referencia a un único load module reubicable con todas las referencias relativas de cada modulo por separado.

¿Como resuelve el link editor la generación de un único load module en el link estático? Primero crea una tabla con todos los módulos objetos y sus respectivas longitudes. Luego declara una dirección base a cada modulo (arranca en 100 porque primero tengo un header)



Una vez que genera esta tabla, el link editor va a buscar todas las referencias a memoria y debera sumarle **una constante de reubicación** igual a la dirección de inicio del módulo objeto. Esto es

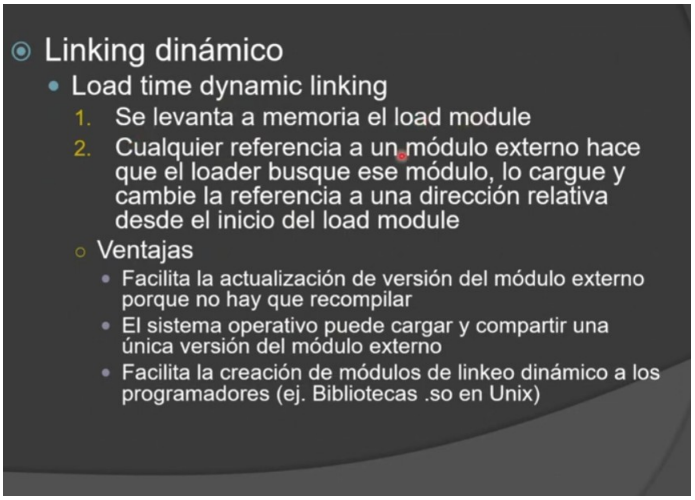
necesario ya que saltar a una instrucción cuando había un módulo solo no es lo mismo que hacerlo cuando esta unido a todos los demás. Para hacer esto utiliza el diccionario de reubicabilidad.

Finalmente, el link editor resuelve las referencias a las instrucciones u otros procedimientos externos e inserta su correspendiente dirección en mi tabla de direcciones externas en mi código objeto.

El link dinámico yo puedo generar un ejecutable que no tenga todo el código binario para correr, si no que puede aparecer luego, por ejemplo en tiempo de carga o en tiempo de ejecución. Es decir, en principio tengo referencias externas sin resolver que el loader identifica.

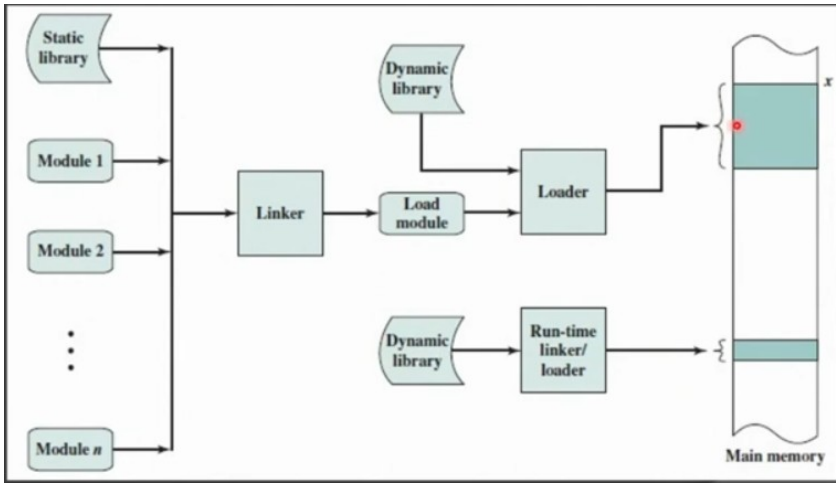
Linking dinámico en tiempo de carga

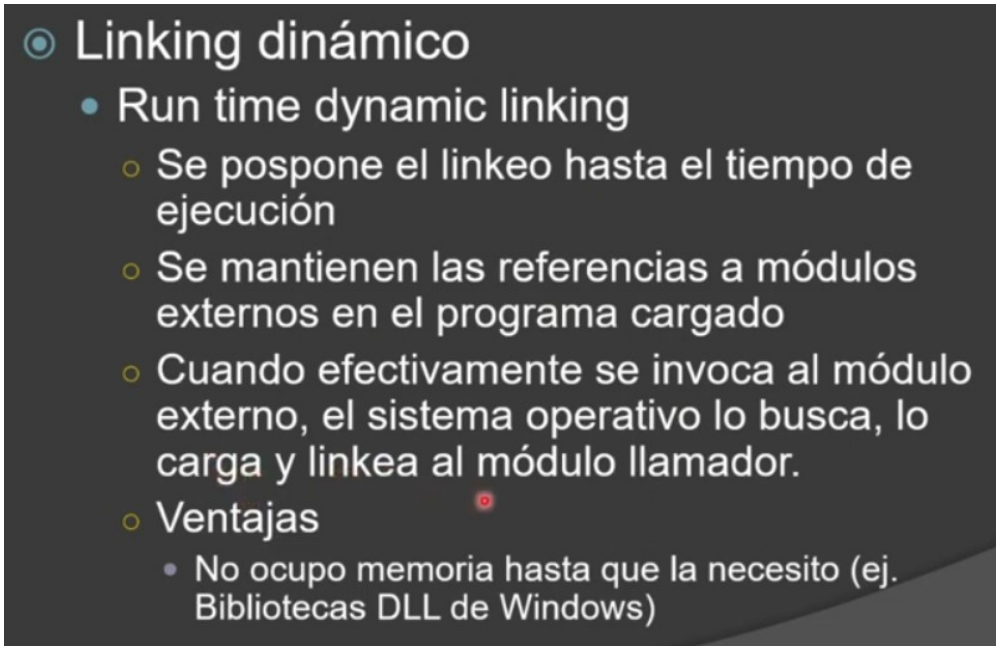
En tiempo de carga lo que hace el loader es combinar aquello generado por el load module y la librería dinámica, de modo que cuando se hace referencia a los módulos estáticos a alguna librería externa, cuando sea el tiempo de carga por ejemplo sabrá donde ubicar dicho llamado externo gracias a la referencia que tiene dentro del header de cada código objeto:



Linking dinámico en tiempo de ejecución

Acá el ejecutable tiene una referencia externa pero ni siquiera se carga al loader. Esta referenciado de manera que recién sobre la memoria es que se carga la información. El sistema operativo es el encargado de reconocer dicha marca y va a ir a buscar a una biblioteca de modulos dinámicos y lo va a ubicar dentro de la memoria ram.





Proceso de Loading

Este proceso hoy en dia está generalizado en el loading dinámico en tiempo de ejecución. Este, cuando el loader carga el programa a memoria lo deja con sus correspondientes direcciones relativas y lo que ocurre es una traducción por linea y constantemente de las direcciones reales en absolutas. Es decir, en tiempo de ejecución, un componente de hardware (MMU – Memory Management Unit) se encarga de hacer la traducción de lo relativo a lo absoluto.

Después existen otras implementaciones donde todas las direcciones no se cargan relativas a un cero, si no que se hace a un registro de la máquina (loading por registro base). Al registro base se le asigna el valor de donde está cargado el programa y a partir de este todas las direcciones quedan resueltas tal que las direcciones relativas al inicio se mantendrían.

El loading absoluto y reubicable son mas obsoletos. El primero, las direcciones reales están prefijadas desde un inicio. El segundo cambia las direcciones en el momento de carga pero no era tan estable.