8. Compilación, carga y depuración de programas

Elena García-Morato, Felipe Ortega, Gorka Guardiola, Enrique Soriano

GSyC, ETSIT. URJC.

Laboratorio de Sistemas (LSIS)

24 abril, 2023

Contenidos

8.1 ELF

ELF

- Executable and Linkable Format.
- Format binario de fichero para ejecución nativa en Linux.

```
$ file ‘which ls‘
/bin/ls: ELF 64-bit LSB shared object, x86-64,
  version 1 (SYSV), dynamically linked,
  interpreter /lib64/ld-linux-x86-64.so.2,
  for GNU/Linux 3.2.0,
  BuildID[sha1]=bf40cb84e7815de09fa792a097061886933e56fa,
  stripped
```

8.2 Compilación y enlazado

Librerías

- A veces un programa utiliza funciones ya programadas.
- Se llaman librerías (o bibliotecas en su traducción estricta de *libraries*).
- Las funciones, variables, etc. de estas librerías pueden ir dentro del binario final (enlazado estático).
- Como alternativa, pueden ir en un fichero binario separado (enlazado dinámico, librerías dinámicas).
- En realidad, tiene que ver con cuándo se resuelven los símbolos, que veremos a continuación.

Programas con varios ficheros

- Muchas veces un programa está compuesto por varios ficheros, que se pueden organizar en *módulos* o *paquetes* (dependiendo, entre otros factores, del lenguaje de programación que utilicemos).
- En ese caso, es parecido a una librería, pero sin empaquetar.
  - Las librerías suelen estructurarse para que se pueda importar su código directamente y facilitar su incorporación a un proyecto software.
- Los programas que nosotros desarrollemos deberían ir empaquetados o, al menos, preparados para facilitar su instalación/utilización por otras personas que quieran usarlo.
- Ejemplo: la herramienta Setuptools de Python.
  - Permite empaquetar el proyecto y distribuirlo, por ejemplo, subiéndolo a Python Package Index (PyPI).

Compilación + enlazado

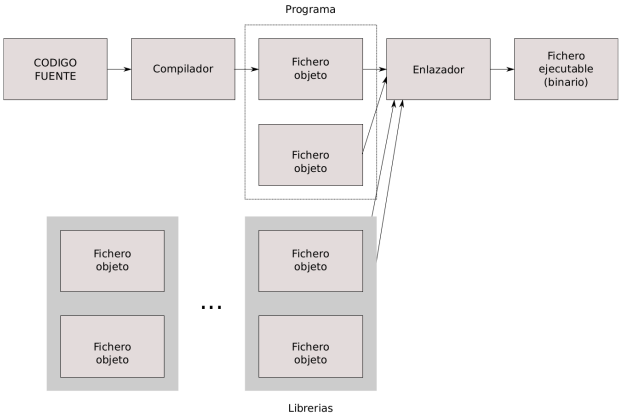


Figura 1: Esquema del proceso de compilación y enlazado de un proyecto que incorpora varias bibliotecas.

Compilación + enlazado (ejemplo C)

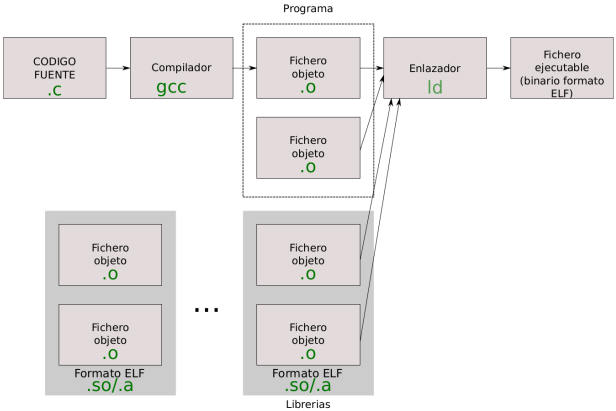


Figura 2: Ejemplo del proceso de compilación y enlazado de un proyecto que incorpora varias bibliotecas.

Código relocalizable

- El código máquina que genera el compilador, puede ser *relocalizable* o no.
- Si es relocalizable, todos los saltos, direcciones de memoria, etc. son relativos.
  - El programa se puede poner en varios sitios en memoria y funcionaría.
- En el siguiente ejemplo, la primera instrucción no es relocalizable, la segunda sí (posición relativa a la dirección de comienzo del programa).

```
JMP 0x344c1cd3
vs
JMP START+0x34
```

Abstracción de proceso

- El sistema operativo le da al programa que ejecuta memoria que empieza en 0.
- Todos los programas creen que tienen toda la memoria (memoria virtual, *hardware* + Sistema operativo).
- Cuando el programa esté ejecutando (proceso), tiene que saltar a direcciones concretas.
  - Imaginemos que tenemos una llamada a función.
  - En algún momento esta llamada a función pasa de ser simbólica `CALL print` a concreta `CALL 0x234c2d34`.
  - Esto se llama resolver el símbolo y pasa con todos los identificadores.

Compilador

- El objetivo del compilador es traducir del lenguaje de alto nivel (C, go, C++) a código máquina.
- Para ello construye un árbol de sintaxis (AST) y luego lo traduce.

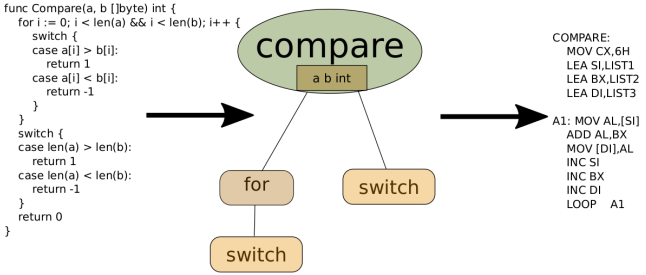


Figura 3: Proceso de un compilador C para transformar el programa original a alto nivel en instrucciones máquina.

Compilador

- El objetivo del compilador es traducir del lenguaje de alto nivel (C, go, C++) a código máquina.
- Para ello construye un árbol de sintaxis (AST) y luego lo traduce.

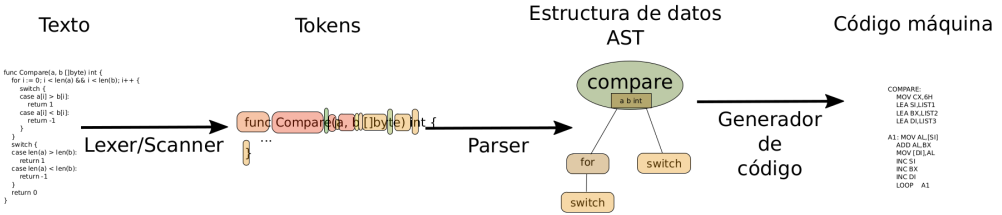
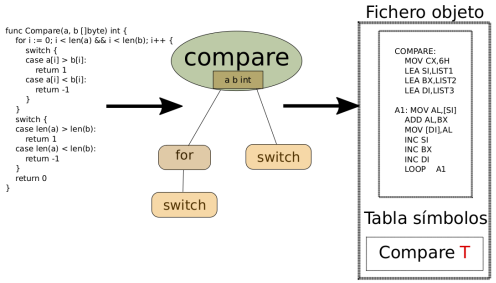


Figura 4: Proceso de un compilador C para transformar el programa original a alto nivel en instrucciones máquina.

Compilador

- El compilador resuelve todos los símbolos definidos en el fichero.
- Para todos los símbolos, crea una tabla
- Los símbolos resueltos para depurar y dar mensajes de error; los no resueltos para más adelante en el enlazado (con las librerías) resolverlos



Compilador

- El fichero que genera el compilador es un fichero objeto.
- Código relocalizable.
- Posiblemente con símbolos sin resolver.

Enlazado

- Despues del enlazado, queda un fichero ejecutable (por ejemplo Elf).
- El enlazado puede suceder en:
  - Tiempo de compilación: enlazado estático, el fichero ejecutable tiene todos los símbolos resueltos y es autosuficiente.
  - Tiempo de ejecución: enlazado dinámico, el fichero ejecutable tiene símbolos que se resuelven al ejecutar, depende de librerías dinámicas.
- En ambos casos puede haber tabla de símbolos con símbolos resueltos o no, para depurar.

Comandos

- `nm` imprime la tabla de símbolos.
- `strip` quita la tabla de símbolos (hace más pequeño el binario, más dificil la depuración).

Carga

- Para ejecutar un binario, se copia a memoria y se salta al punto de entrada (se pone el registro PC en el punto de entrada), se prepara la pila, etc.
- Esto se llama carga, y si el binario requiere estar en un punto de la memoria concreta (código no relocalizable), que es lo normal, hay que copiarlo ahí.
- Si tiene enlazado dinámico hay que cargar también las librerías dinámicas y resolver los símbolos (esto lo hacen `ld.so` y `ld-linux.so` en colaboración con el *kernel*, ver `ld.so(8)`).
- Se puede ir copiando el binario cuando se necesita, esto se llama *carga en demanda*.

8.3 Librerías

Librerías dinámicas y objetos

- Las librerías dinámicas y estáticas y los objetos están en un formato especificado por `Elf` en Linux (es parte del mismo estándar)
- La tabla de símbolos están en Dwarf, un formato asociado a Elf.
- Las librerías dinámicas son normalmente ficheros acabados en `.so` y las estáticas acabadas en `.a`.

Ubicación de las librerías dinámicas

- Al compilar, pueden tener un path absoluto, si no (simplificado, más detalles en `ld.so(8)`):
  - Se busca en la variable de entorno `LD_LIBRARY_PATH`.
  - Luego en la cache `/etc/ld.so.cache`.
  - Si no, `/lib`, `/usr/lib` (en algunos `/lib64`, `/usr/lib64`).
- Para saber las dependencias, `readelf -d fichero`.
- La configuración de la caché y los enlaces simbólicos de las librerías dinámicas las hace `ldconfig`.
- La configuración de `ldconfig` está en `/etc/ld.so.conf` y `/etc/ld.so.conf.d`.

8.4 Depuración

Tareas de depuración

Existen varios tipos de errores.

- Compilación.
- Ejecución.
- Funcionales.
- Problemas de rendimiento, liberación de recursos.

Tareas de depuración

- Experimentalidad vs. idea mental incorrecta.
- Cuando depuramos estamos intentando ver qué hace el código.
- Puramente experimental (como la física).

Estrategias de depuración

- Volcar el estado del programa (print, trazas).
- Antes y después de que pase a ser incorrecto.
- Ir dividiendo en 2 el código, buena convergencia con el número de líneas ($O(ln(n))$).
- Instrumentar las estructuras de datos, que impriman su estado.
- Tener niveles de depuración habilitables, (*verbose*).
- También podemos usar herramientas profesionales de depuración que incorpore nuestra IDE favorita.
- Es muy recomendable utilizar en nuestro proyecto un *logger*, una librería que crea archivos de bitácora de manera profesional.

El depurador (armas pesadas)

- Programa que me permite parar el programa e inspeccionarlo (ejemplo gdb).
- O inspeccionar un core (volcado de memoria de programa muerto).
- Permite desensamblarlo, etc.

El depurador (armas pesadas)

- Se usa sólo si no me queda más remedio.
- Desensamblado.
- Ejecución paso a paso (mala idea, $O(n)$, consideraciones de tiempo (*multithread*, red. . .).
- `Breakpoint`: que pare cuando llege a un punto.
- `Watchpoint`: que pare cuando se acceda a una variable.

8.5 Profiling

Profiling

- Herramienta: `profiler`, medir el uso de recursos, memoria, CPU, búsqueda de puntos calientes (hot spots).
- También se puede hacer a mano midiendo tiempo, contando pasos por un sitio o asignación y liberación de recursos.
- Existen varios tipos:
  - Traza estadística (cada cierto tiempo mide donde está).
  - Inyecta contadores de tiempo en la funciones, contadores de asignación/liberación de recursos en memoria...
  - Contadores hardware.
- Lo que no se ha medido no se sabe bien (siempre hay sorpresas), medir antes de optimizar

8.6 Análisis estático y dinámico

Análisis estático

- Herramienta: `linter`, para encontrar errores y problemas automáticamente.
- Hace análisis estático del código.
- Patrones incorrectos o fácilmente erróneos de uso.
- Peligro de falsos positivos.
- https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

Análisis dinámico: Valgrind

- Ayuda para encontrar errores y problemas automáticamente.
- Hace análisis dinámico del código en ejecución (instrumentado o sin instrumentar).
- *Leaks*, corrupción de memoria, patrones incorrectos de uso de los recursos.
- `http://valgrind.org`.

Para saber más

- El capítulo 16 de [3] explica de forma detallada el proceso completo de compilación de software a partir de código fuente en C, incluyendo algunas herramientas adicionales que no hemos mencionado como *GNU Autoconf*.

Referencias I

[1] M. Hausenblas. *Learning Modern Linux*. O'Reilly Media, abr. de 2022.

[2] E. Nemeth y col. *Unix and Linux System Administration Handbook*. 4ª ed. Pearson, 2010.

[3] B. Ward. *How Linux Works: What Every Superuser Should Know*. 3ª ed. No Starch Press, abr. de 2021.